Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets

Samuel Kounev, *Member*, *IEEE Computer Society*

Abstract—Performance models are used increasingly throughout the phases of the software engineering lifecycle of distributed component-based systems. However, as systems grow in size and complexity, building models that accurately capture the different aspects of their behavior becomes a more and more challenging task. In this paper, we present a novel case study of a realistic distributed component-based system, showing how Queueing Petri Net models can be exploited as a powerful performance prediction tool in the software engineering process. A detailed system model is built in a step-by-step fashion, validated, and then used to evaluate the system performance and scalability. Along with the case study, a practical performance modeling methodology is presented which helps to construct models that accurately reflect the system performance and scalability characteristics. Taking advantage of the modeling power and expressiveness of Queueing Petri Nets, our approach makes it possible to model the system at a higher degree of accuracy, providing a number of important benefits.

Index Terms—Performance modeling and prediction, software verification, performance evaluation, distributed systems.

1 Introduction

Distributed component-based systems (DCS) are becoming increasingly ubiquitous as enabling technology for modern enterprise applications. In the face of globalization and ever increasing competition, *Quality of Service (QoS)* requirements on such systems, like performance, availability, and reliability, are of crucial importance. Businesses must ensure that the systems they operate not only provide all relevant services, but also meet the performance expectations of their customers. To avoid the pitfalls of inadequate QoS, it is important to analyze the expected performance characteristics of systems during all phases of their life cycle. The methods used to do this are part of the discipline called *Performance Engineering* [1]. Performance engineering helps to estimate the level of performance a system can achieve and provides recommendations to realize the optimal performance level [2]. The latter is done by means of system models that are used to predict the performance of the system under the expected workload. However, building models that accurately capture the different aspects of system behavior is an extremely challenging task when applied to large and complex real-world systems.

In [3] and [4], we presented two practical performance modeling case studies which demonstrated the difficulties that arise when trying to model a realistic DCS and predict its performance. In the first case study, we used conventional Queueing Network (QN) models to evaluate the performance of a large J2EE[1] application. While the models were shown to capture the hardware contention aspects of system behavior well, due to the limited expressiveness of QNs, it was not possible to *accurately* model asynchronous processing and software contention aspects. Moreover, the available model analysis techniques failed to provide reliable response time predictions when increasing the workload intensity. In the second case study, we modeled a similar J2EE application, but using Queueing Petri Net (QPN) models instead of QNs. Exploiting the modeling power and expressiveness of QPN models, we were able to accurately capture both hardware and software aspects of system behavior. However, the resulting models were by far too large to be analyzable using the tools and techniques for QPN analysis available at that time. Therefore, we had to simplify the models, restricting them to a small part of the application in order to avoid state space explosion. We demonstrated that QPN models lend themselves very well to modeling DCS, however, new solution techniques and tools for QPN analysis were needed to enable us to solve models of realistic size and complexity. In [5], we addressed this issue by providing a scalable and reliable QPN analysis technique, circumventing the state space explosion problem. The technique, based on discrete event simulation, was implemented as part of a new simulation tool for QPNs, called SimQPN. The latter was subjected to a rigorous experimental analysis and proved to provide very accurate and stable point and interval estimates of performance metrics. Thus, using SimQPN, it was now

• *The author is with the Department of Computer Science, Darmstadt University of Technology, 64289 Darmstadt, Germany, and the University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK.*
E-mail: skounev@acm.org, skounev@informatik.tu-darmstadt.de.

Manuscript received 4 Jan. 2005; revised 8 May 2006; accepted 17 May 2006; published online 9 Aug. 2006.
Recommended for acceptance by R. Schlichting.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0001-0105.

1. The Java 2 Enterprise Edition Platform (J2EE) defines an industry standard for implementing middleware platforms for DCS. J2EE-based platforms are currently the technology of choice for building DCS.

possible to analyze QPN models of realistic size and complexity, taking advantage of the modeling power and expressiveness of the QPN paradigm.

In this paper, we present a novel case study of a realistic state-of-the-art DCS, showing how the QPN modeling formalism can be exploited to its full potential as a performance prediction tool in the software engineering process. Along with the case study, we present a practical performance modeling methodology which helps to construct models of DCS that *accurately* reflect their performance and scalability characteristics. The methodology is based on existing work in software performance engineering by Menascé et al. [6], [7], [8], [9], [2], Smith and Williams [10], [11], [12], and Woodside et al. [13], [14], [15], [16], [17]. However, a major new aspect is the use of QPN models as opposed to conventional QN models. As mentioned above, QPN models are more sophisticated than QN models and enjoy greater modeling power and expressiveness [18], [19], [4]. This provides a number of important benefits since it enables us to model the system at a higher degree of accuracy.

The paper starts with a brief overview of the modeling methodology and, after that, the case study is presented which is the main contribution. The system studied is a deployment of the industry-standard SPECjAppServer2004[2] benchmark for J2EE application servers. The latter implements a *new* enhanced workload that is substantially more complex and comprehensive than previous versions of the SPECjAppServer [20], [21]. It models a complete end-to-end application that is designed to be representative of today's real-life DCS. The case study considers a deployment of the benchmark application and shows how to build a detailed QPN model of the latter, validate it, and then use it to evaluate the system performance and scalability. The reader is walked step-by-step through the modeling and evaluation process. A number of deployment configurations and workload scenarios are considered. In each case, the model is analyzed by means of simulation using SimQPN—our simulation tool for QPNs [5]. In order to validate the approach, the model predictions are compared against measurements on the real system. The case study complements and extends our work in [3] and [4] along the following dimensions:

- An application, larger, more complex, and much more representative of today's DCS than the applications considered previously, is studied.
- This time the system is modeled in its entirety, resulting in a more detailed and comprehensive model.
- Two important aspects of system behavior, not considered previously, are now modeled explicitly: composite transactions and asynchronous processing.[3]
- The case study introduces QPN departure disciplines—an easy to use yet powerful feature allowing us to control the order in which tokens leave a QPN place.
- The performance of the system is predicted accurately for workload scenarios under realistic load conditions with up to 540 concurrent jobs, including scenarios with software contention.

Thanks to the increased modeling accuracy and representativeness, the models considered in this paper demonstrate much better predictive power and scalability than was achieved in our previous work. The case study we present here is the first comprehensive validation of our modeling approach on a *complete end-to-end* DCS, representative of today's real-life systems.

The paper is organized as follows: In Section 2, a brief overview of the performance modeling methodology that is used in the case study is given. It is shown how QPNs can be exploited to model the different aspects of system behavior in an accurate manner. Following this, Section 3 presents our case study of SPECjAppServer2004 showing how each step of the modeling and evaluation process is applied in practice. The approach is validated by comparing model predictions against measurements on the real system. Finally, the paper is summarized in Section 4. For readers not familiar with Queueing Petri Nets, a brief introduction is included in the Appendix.

2 Performance Modeling Methodology

We begin by giving an overview of the performance modeling methodology that is used in the case study. As already mentioned, this methodology is based on existing work in software performance engineering by Menascé et al. [6], [7], [8], [9], [2], Smith and Williams [10], [11], [12], and Woodside et al. [13], [14], [15], [16], [17]. A major new aspect, however, is that we use QPN models as opposed to conventional QN models. This is an important extension since it enables us to model the system at a higher degree of accuracy. As shown in [19], [22], QPNs have greater expressive power than QNs, extended QNs, and stochastic Petri nets. Taking advantage of this, our approach provides several important benefits. First of all, QPN models allow the integration of hardware and software aspects of system behavior and lend themselves very well to modeling DCS [23], [4]. In addition to hardware contention and scheduling strategies, using QPNs one can easily model software contention, simultaneous resource possession, synchronization, blocking, and asynchronous processing. These aspects of system behavior, which are typical for modern DCS, are difficult to model *accurately* using conventional QN models. Second, by restricting ourselves to QPN models, we can exploit the knowledge of their structure and behavior for fast and efficient analysis using simulation [5]. This enables us to solve models of large and complex DCS and ensures that our approach scales to realistic systems. Finally, many efficient qualitative analysis techniques from Petri net theory can be extended to QPNs and used to combine qualitative and quantitative system analysis [18].

2. SPECjAppServer is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjAppServer2004 results or findings in this publication have not been reviewed or accepted by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official Web site for SPECjAppServer2004 is located at http://www.spec.org/osg/jAppServer2004.

3. Note that, even though, in [3], an approach for approximate modeling of asynchronous processing was presented, this approach was rather primitive and had some significant limitations.

2.1 Methodology Overview

The methodology we employ includes seven steps.

2.1.1 Establish Performance Modeling Objectives

The first step is to set some concrete goals for the performance modeling effort. The latter should be stated in a simple and precise manner.

2.1.2 Characterize the System in Its Current State

The product of this step is a specification that includes detailed information on the system design and topology, the hardware and software components it is comprised of, the communication and network infrastructure, etc.

2.1.3 Characterize the Workload

In the third step, the workload of the system under study is described in a qualitative and quantitative manner. This includes five major steps [6], [24]:

1. *Identify the Basic Components of the Workload.* Basic component refers to a generic unit of work that arrives at the system from an external source, for example, HTTP request, remote procedure call, or database transaction [8].
2. *Partition Basic Components into Workload Classes.* In order to improve the representativeness of the workload model and increase its predictive power, the basic components are partitioned into classes (called *workload classes*) that have similar characteristics [25], [8].
3. *Identify the System Components and Resources Used by Each Workload Class.* This includes both hardware and software components, as well as active and passive resources [6].
4. *Describe the Intercomponent Interactions and Processing Steps.* Different notations may be exploited here, for example, Client/Server Interaction Diagrams (CSID) [7], Communication-Processing Delay Diagrams [8], Execution Graphs [10], as well as conventional UML Sequence and Activity Diagrams [26].
5. *Characterize Workload Classes in Terms of Service Demands and Workload Intensity.* Most techniques for obtaining service demand parameters involve running the system or components thereof and taking measurements. For detailed information on measurement techniques, refer to [2], [8], [6], [27]. Some techniques are also available that can be used to estimate the service demands in the early stages of development before the system is available for testing [28].

2.1.4 Develop a Performance Model

In this step, a performance model is developed that represents the different components of the system and its workload and captures the main factors affecting its performance. In our approach, we use QPN models, taking advantage of their increased expressiveness to improve the model representativeness and predictive power.

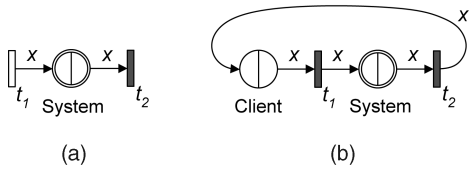


Fig. 1. Modeling request/transaction arrivals and departures. (a) Open classes. (b) Closed classes.

2.1.5 Validate, Refine, and/or Calibrate the Model

The model is said to be *valid* if the performance metrics predicted by the model match the measurements on the real system within a certain acceptable margin of error [2]. If this is not the case, the model must be refined or calibrated to more accurately reflect the system and workload modeled. For a detailed discussion of model validation and calibration techniques, refer to [29], [6].

2.1.6 Use Model to Predict System Performance

In this step, the validated performance model is used to predict the performance of the system for the deployment configurations and workload scenarios targeted for analysis. The latter are derived from the modeling objectives.

2.1.7 Analyze Results and Address Modeling Objectives

Finally, in the last step of the methodology, the results from the model predictions are analyzed and used to address the goals set in the beginning of the modeling study.

2.2 Modeling Using Queueing Petri Nets

Now that we have discussed the modeling methodology in general, we briefly describe our approach for building QPN models of DCS. The modeling process includes the following steps:

2.2.1 Model the System Components and Resources

The first step is to map the system components and resources (hardware and software) to respective QPN model constructs. Active resources are usually modeled using queueing places. Passive resources such as threads, processes, database connections, and locks are normally modeled using tokens inside ordinary places.

2.2.2 Model the Basic Components of the Workload

The basic components of the workload are modeled using tokens, exploiting different colors to distinguish between workload classes. Some additional QPN constructs are needed to model the way transactions get started and completed in the system or, equivalently, the way requests arrive and depart from the system. Fig. 1 illustrates how this is done for open and closed workload classes.

For open workload classes, two transitions, one timed (t_1) and one immediate (t_2), are needed. Whenever t_1 fires, it simply creates a new transaction token and deposits it into the system, represented in the figure using a subnet place (a nested QPN). The firing delay is chosen according to the desired transaction injection rate. The immediate transition is used to destroy tokens of completed transactions. For closed workload classes, a queueing place (*Client*) with IS queue and two immediate transitions are used. Two cases

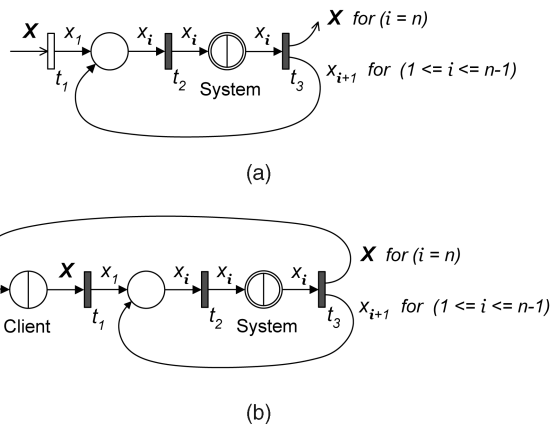


Fig. 2. Modeling composite transactions. (a) Open classes. (b) Closed classes.

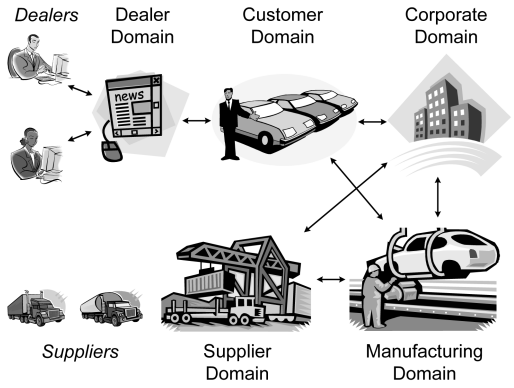


Fig. 3. SPECjAppServer2004 business model.

are distinguished, based on whether workload intensity is specified as an average number of transactions being processed concurrently or it is specified as a number of active clients/terminals that start transactions. In the first case, tokens in the Client place are served immediately and its initial token population determines the number of concurrent transactions in the system.[4] In the second case, the Client place has service time equal to the average client think time and its initial token population determines the number of active clients/terminals starting transactions.

2.2.3 Model the Intercomponent Interactions and Processing Steps

The interactions between system components are normally modeled using transitions connecting the QPN places corresponding to the respective components. Transitions are used to model the flow of control from one system component to another when processing a transaction.

For composite transactions, the individual processing steps (subtransactions) can also be modeled using tokens. One way to do this is to use a separate token color for every subtransaction. This is illustrated in Fig. 2 for open and closed workload classes, respectively. An upper case X token is used to represent a composite transaction. The individual subtransactions of the latter are represented using lower case x tokens, where x_i stands for the ith subtransaction. When the transaction is started (by firing transition t_1), a token x_1 representing the first subtransaction is created and deposited into the system. After the subtransaction is completed, its token is destroyed (by transition t_3) and a token representing the next processing step x_{i+1} is created and deposited into the system. This process continues until the last subtransaction (modeled as token x_n) has been processed.

2.2.4 Parameterize the Model

The last step of the modeling process is to provide values for the following parameters: initial token population of places, service times of tokens at the queues of queueing places, firing weights of immediate transitions, and firing delays of timed transitions.

4. In fact, in this case, the Client place is not necessarily needed and one can simplify the model by removing the Client place and transition t_1 and using transition t_2 as a short-circuit.

3 Case Study: Modeling SPECjAppServer2004

Now that we have discussed our modeling approach in general, we present our case study with SPECjAppServer2004. The case study demonstrates how the QPN modeling paradigm can be exploited to its full potential as a performance prediction tool for DCS. An application, larger, more complex, and much more representative of today's DCS than the applications considered in our previous work ([3] and [4]), is studied. Moreover, while, in our previous case studies, some significant simplifications had to be made in order to avoid the largeness problem, this time the application studied is modeled in its entirety, resulting in a much more detailed and comprehensive model.

3.1 The SPECjAppServer2004 Benchmark

SPECjAppServer2004 is a new industry-standard benchmark for measuring the performance and scalability of J2EE hardware and software platforms. It implements a representative workload that exercises all major services of the J2EE platform in a complete *end-to-end* application scenario. The SPECjAppServer2004 workload has been specifically modeled after an automobile manufacturer whose main customers are automobile dealers [30]. Dealers use a Web-based user interface to browse an automobile catalog, purchase automobiles, sell automobiles, and track their inventory. As depicted in Fig. 3, SPECjAppServer2004's business model is comprised of five domains: the customer domain dealing with customer orders and interactions, the dealer domain offering Web-based interface to the services in the customer domain, the manufacturing domain performing "just in time" manufacturing operations, the supplier domain handling interactions with external suppliers, and the corporate domain managing all dealer, supplier, and automobile information.

The customer domain hosts an *order entry application* that provides some typical online ordering functionality. Orders for more than 100 automobiles are called *large orders*. The dealer domain hosts a Web application (called *dealer application*) that provides a Web-based interface to the

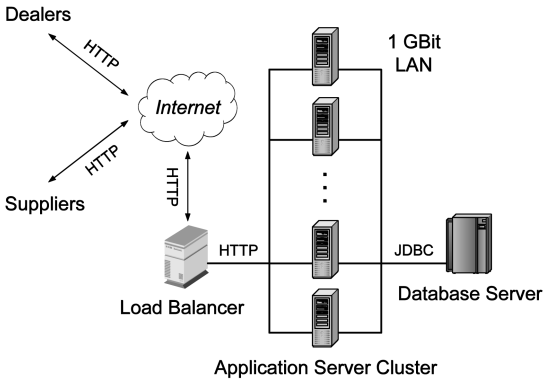


Fig. 4. Deployment environment.

services in the customer domain. The manufacturing domain hosts a *manufacturing application* that models the activity of production lines in an automobile manufacturing plant. There are two types of production lines, planned lines and large order lines. Planned lines run on schedule and produce a predefined number of automobiles. 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 moves along three 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 (333 ms) at each station. Once the work order is complete, it is marked as completed and inventory is updated. When the inventory of parts gets depleted, suppliers need to be located and purchase orders need to be sent out. This is done by contacting the supplier domain responsible for interactions with external suppliers.

3.2 Motivation

Consider an automobile manufacturing company that wants to use e-business technology to support its order-inventory, supply-chain, and manufacturing operations. The company has decided to employ the J2EE platform and is in the process of developing a J2EE application. Let us assume that the first prototype of this application is SPECjAppServer2004 and that the company is testing the application in the deployment environment depicted in Fig. 4. 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 servers in the WebLogic cluster are identical and that, initially, only two servers are available. The company is now about to conduct a performance modeling study of their system in order to evaluate its performance and scalability.

3.3 Establish Performance Modeling Objectives

Let us assume that, under normal operating conditions, the company expects to have 72 concurrent dealer clients (40 Browse, 16 Purchase, and 16 Manage) and 50 planned production lines. During peak conditions, 152 concurrent dealer clients (100 Browse, 26 Purchase, and 26 Manage) are expected and the number of planned production lines could increase up to 100. Moreover, the workload is forecast to grow by 300 percent over the next five years. The average *dealer think time* is 5 seconds, i.e., the time a dealer "thinks" after receiving a response from the system before sending a new request. On average, 10 percent of all orders placed are assumed to be large orders. The average delay after completing a work order at a planned production line before starting a new one is 10 seconds. Note that all of these numbers were chosen arbitrarily in order to make our motivating scenario more specific. Based on these assumptions, the following concrete goals are established:

- Predict the performance of the system under normal operating conditions with four and six WebLogic servers, respectively. What would be the average throughput and response time of dealer transactions and work orders? What would be the CPU utilization of the servers?
- Determine if six WebLogic servers would be enough to ensure that the average response times of business transactions do not exceed half a second during peak conditions.
- Predict how much system performance would improve if the load balancer is upgraded with a slightly faster CPU.
- Study the scalability of the system as the workload increases and additional WebLogic servers are added.
- Determine which servers would be most utilized under heavy load and investigate if they are potential bottlenecks.

TABLE 1
System Component Details

Component	Description
Load Balancer	Commercial HTTP Load Balancer 1 x AMD Athlon XP2000+ CPU 1 GB RAM, SuSE Linux 8
App. Server Cluster Nodes	WebLogic 8.1 Server 1 x AMD Athlon XP2000+ CPU 1 GB RAM, SuSE Linux 8
Database Server	Oracle 9i Server 2 x AMD Athlon MP2000+ CPU 2 GB RAM, SuSE Linux 8
Local Area Network	1 GBit Switched Ethernet

3.4 Characterize the System in Its Current State

As shown in Fig. 4, the system we are considering has a two-tier hardware architecture consisting of an application server tier and a database server tier. Incoming requests are evenly distributed across the nodes in the application server cluster. For HTTP requests, this is achieved using a software load balancer running on a dedicated machine. For RMI requests, this is done transparently by the EJB client stubs. Table 1 describes the system components in terms of the hardware and software platforms used. This information is enough for the purposes of our study.

3.5 Characterize the Workload

3.5.1 Identify the Basic Components of the Workload

As discussed in Section 3.1, the SPECjAppServer2004 benchmark application is made up of three major subapplications—the dealer application, the order entry

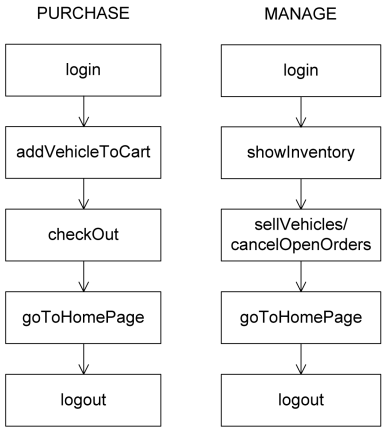


Fig. 5. Execution graphs for Purchase and Manage.

application, and the manufacturing application. The dealer and order entry applications process business transactions of three types—Browse, Purchase, and Manage. Hereafter, the latter are referred to as *dealer transactions*. The manufacturing application, on the other hand, is running production lines which process work orders. Thus, the SPECjAppServer2004 workload is composed of two basic components: dealer transactions and work orders.

Each dealer transaction emulates a client session comprised of multiple round-trips to the server. For every round-trip, there is a separate HTTP request, which can be seen as a subtransaction. A more fine-grained approach to model the workload would be to define the individual HTTP requests as basic components. However, this would unnecessarily complicate the workload model since we are interested in the performance of dealer transactions as a whole and not the performance of their individual subtransactions. The same reasoning applies to work orders because each work order is comprised of multiple JTA transactions initiated with separate RMI calls. This is a typical example of how the level of detail in the modeling process is decided based on the modeling objectives.

3.5.2 Partition Basic Components into Workload Classes

There are three types of dealer transactions and, since we are interested in their individual behavior, we model them using separate workload classes. Work orders, on the other hand, can be divided into two types based on whether they are processed on a planned or large order line. Planned lines run on schedule and complete a predefined number of work orders per unit of time. In contrast, large order lines run only when a large order arrives in the customer domain. Each large order generates a separate work order processed *asynchronously* on a dedicated large order line. Thus, work orders originating from large orders are different from ordinary work orders in terms of the way their processing is initiated and in terms of their resource usage. To distinguish between the two types of work orders, they are modeled using two separate workload classes: *WorkOrder* (for ordinary work orders) and *LargeOrder* (for work orders generated by large orders). Altogether, we end up with five workload classes: Browse, Purchase, Manage, WorkOrder, and LargeOrder.

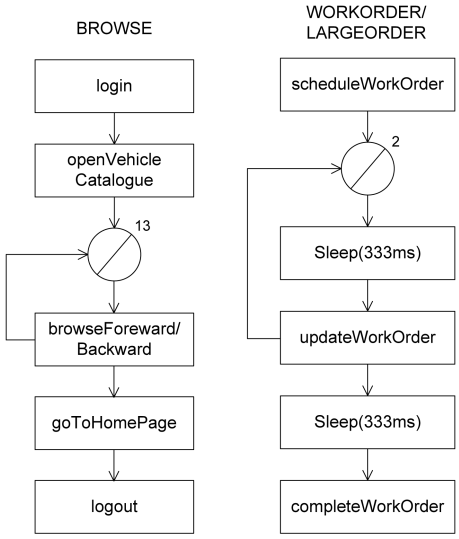


Fig. 6. Execution graphs for Browse, WorkOrder, and LargeOrder.

3.5.3 Identify the System Components and Resources Used by Each Workload Class

The following hardware resources are used by dealer transactions:

- The CPU of the load balancer machine (LB-C).
- The CPU of an application server in the cluster (AS-C).
- The CPUs of the database server (DB-C).
- The disk drive of the database server (DB-D).
- The Local Area Network (LAN).

WorkOrders and LargeOrders use the same resources with the exception of the first one, since their processing is driven through direct RMI calls to the EJBs in the WebLogic cluster, bypassing the HTTP load balancer. As far as software resources are concerned, all workload classes use the WebLogic servers and the Oracle DBMS. Dealer transactions additionally use the HTTP load balancer, which is running on a dedicated machine.

3.5.4 Describe the Intercomponent Interactions and Processing Steps for Each Workload Class

All five of the workload classes identified represent composite transactions. Fig. 5 and Fig. 6 use execution graphs to illustrate the subtransactions (processing steps) of transactions from the different workload classes. For every subtransaction (represented as a rectangle), multiple system components are involved and they interact to perform the respective operation. The intercomponent interactions and flow of control during the processing of subtransactions are depicted in Fig. 7 by means of client/server interaction diagrams. Directed arcs show the flow of control from one node to the next during execution. Depending on the path followed, different execution scenarios are possible. For example, for dealer subtransactions, two scenarios are possible depending on whether the database needs to be accessed or not. Dealer subtransactions that do not access the database (e.g., goToHomePage) follow the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, whereas dealer subtransactions that access the database (e.g., showInventory) follow the path

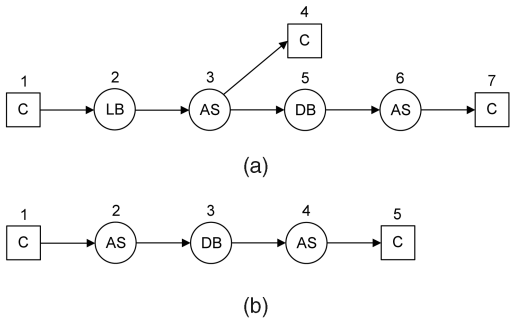


Fig. 7. Client/server interaction diagrams for subtransactions. (a) Subtransactions of Browse, Purchase, and Manage. (b) Subtransactions of WorkOrder and LargeOrder.

$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7$. Since most dealer subtransactions do access the database, for simplicity, it is assumed that all of them follow the second path.

3.5.5 Characterize Workload Classes in Terms of Their Service Demands and Workload Intensity

Since the system is available for testing, the service demands can be determined by injecting load into the system and taking measurements. Note that it is enough to have a single WebLogic server available in order to do this, i.e., it is not required to have a realistic production-like testing environment. For each of the five workload classes, a separate experiment was conducted injecting transactions from the respective class and measuring the utilization of the various system resources. CPU utilization was measured using the `vmstat` utility on Linux. The disk utilization of the database server was measured with the help of the Oracle 9i Intelligent Agent, which proved to have negligible overhead. Service demands were derived using the Service Demand Law [8]. Table 2 reports the service demand parameters for the five workload classes. It was decided to ignore the network since all communications were taking place over 1 GBit LAN and communication times were negligible.

In order to keep the workload model simple, it is assumed that the total service demand of a transaction at a given system resource is spread evenly over its subtransactions. Thus, the service demand of a subtransaction can be estimated by dividing the measured total service demand of the transaction by the number of subtransactions it has. It is also assumed that all service demands are exponentially distributed. Whether these simplifications are acceptable will become clear later when the model is validated. In case the estimation proves to be too inaccurate, one might have to come back and refine the workload model by measuring the service demands of subtransactions individually.

TABLE 2
Workload Service Demand Parameters

Workload Class	LB-C	AS-C	DB-C	DB-D
Browse	42.72ms	130ms	14ms	5ms
Purchase	9.98ms	55ms	16ms	8ms
Manage	9.93ms	59ms	19ms	7ms
WorkOrder	-	34ms	24ms	2ms
LargeOrder	-	92ms	34ms	2ms

TABLE 3
Workload Intensity Parameters

Parameter	Normal Conditions	Peak Conditions
Browse Clients	40	100
Purchase Clients	16	26
Manage Clients	16	26
Planned Lines	50	100
Dealer Think Time	5 sec	5 sec
Mfg Think Time	10 sec	10 sec

Now that the service demands of workload classes have been quantified, the workload intensity must be specified. For each workload class, the number of transactions that contend for system resources must be indicated. The way workload intensity is specified is dictated by the modeling objectives. In our case, workload intensity was defined in terms of the following parameters (see Section 3.3):

- the number of concurrent dealer clients of each type and the average dealer think time and
- the number of planned production lines and the average time they wait after processing a WorkOrder before starting a new one (*manufacturing think time* or *mfg think time*).

With workload intensity specified in this way, all workload classes are automatically modeled as closed. Two scenarios of interest were indicated when discussing the modeling objectives in Section 3.3: operation under normal conditions and operation under peak conditions. The values of the workload intensity parameters for these two scenarios are shown in Table 3. However, the workload had been forecast to grow by 300 percent and another goal of the study was to investigate the scalability of the system as the load increases. Therefore, scenarios with up to 3 times higher workload intensity need to be considered as well.

3.6 Develop a Performance Model

A QPN model of the system under study is now built and then customized to the concrete configurations of interest. We start by discussing the way basic components of the workload are modeled. During workload characterization, five workload classes were identified. All of them represent composite transactions and are modeled using the following token types (colors): "B" for Browse, "P" for Purchase, "M" for Manage, "W" for WorkOrder, and "L" for LargeOrder.

The subtransactions of transactions from the different workload classes were shown in Fig. 5 and Fig. 6. In order to make the performance model more compact, it is assumed that each server used during processing of a subtransaction is visited only once and that the subtransaction receives all of its service demands at the server's resources during that single visit. This simplification is typical for queueing models and has been widely employed. Similarly, during the service of a subtransaction at a server, for each server resource used (e.g., CPUs, disk drives), it is assumed that the latter is visited only one time, receiving the whole service demand of the subtransaction at once. These simplifications make it easier

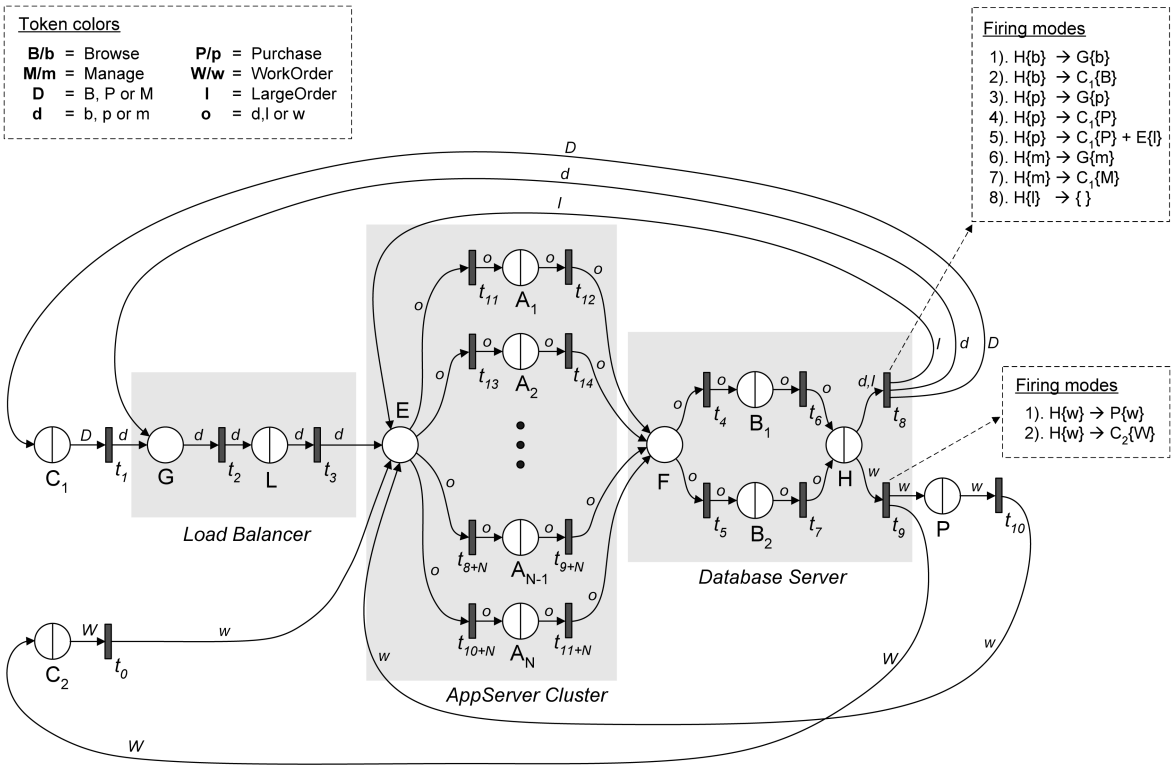


Fig. 8. QPN model of the system.

to model the flow of control during processing of subtransactions. While characterizing the workload service demands in Section 3.5.5, we additionally assumed that the total service demand of a transaction at a given system resource is spread evenly over its subtransactions. This allows us to consider the subtransactions of a given workload class as equivalent in terms of processing behavior and resource consumption. Thus, we can model subtransactions using a single token type (color) per workload class as follows: "b" for Browse, "p" for Purchase, "m" for Manage, "w" for WorkOrder, and "l" for LargeOrder. For the sake of compactness, the following additional notation will be used:

- Symbol "D" will denote a "B," "P," or "M" token, i.e., token representing a dealer transaction.
- Symbol "d" will denote a "b," "p," or "m" token, i.e., token representing a dealer subtransaction.
- Symbol "o" will denote a "b," "p," "m," "w," or "l" token, i.e., token representing a subtransaction of arbitrary type, hereafter called *subtransaction token*.

To further simplify the model, we assume that LargeOrder transactions are executed with a single subtransaction, i.e., their four subtransactions are bundled into a single subtransaction. Thus, the total service demand of a LargeOrder transaction at a given system resource is assumed to be received at once, during a single visit to the resource. The effect of this simplification on the overall system behavior is negligible because large orders constitute only 10 percent of all orders placed, i.e., a relatively small portion of the system workload. Following these lines of thought, one could consider LargeOrder transactions as noncomposite and drop the small "l" tokens. However, in order to keep token definitions uniform across transaction classes, we will keep the small "l" tokens and look at LargeOrder transactions as being composed of a single subtransaction represented by an "l" token.

Following the guidelines for modeling the system components, resources, and intercomponent interactions presented in Section 2.2, we arrive at the model depicted in Fig. 8. We use the notation "$A\{x\} \rightarrow B\{y\}$" to denote a firing mode in which an "x" token is removed from place A and a "y" token is deposited in place B. Similarly, "$A\{x\} \rightarrow \{\}$" means that an "x" token is removed from place A and destroyed without depositing tokens anywhere. Table 4 provides some details on the places used in the model.

All token service times at the queues of the model are assumed to be exponentially distributed. We now examine in detail the life-cycle of tokens in the QPN model. As already discussed, upper case tokens represent transactions, whereas lower case tokens represent subtransactions. In the initial marking, tokens exist only in the depositories of places C_1 and C_2. The initial number of "D" tokens ("B," "P," or "M") in the depository of the former determines the number of concurrent dealer clients, whereas the initial number of "W" tokens in the depository of the latter determines the number of planned production lines running in the manufacturing domain. When a dealer client starts a dealer transaction, transition t_1 is fired, destroying a "D" token from the depository of place C_1 and creating a "d" token in place G, which corresponds to starting the first

TABLE 4
Places Used in the QPN Model

Place	Tokens	Queue Type	Description
C_1	{B,P,M}	$G/M/\infty/IS$	Queueing place used to model concurrent dealer clients conducting dealer transactions. The time tokens spend here corresponds to the dealer think time.
C_2	{W}	$G/M/\infty/IS$	Queueing place used to model planned production lines driving work order processing. The time tokens spend here corresponds to the mfg think time.
G	{b,p,m}	*na*	Ordinary place where dealer subtransaction tokens are created when new subtransactions are started.
L	{b,p,m}	$G/M/1/PS$	Queueing place used to model the CPU of the load balancer machine.
E	{b,p,m,l,w}	*na*	Ordinary place where subtransaction tokens arrive before they are distributed over the application server nodes.
A_i	{b,p,m,l,w}	$G/M/1/PS$	Queueing places used to model the CPUs of the *N* application server nodes.
F	{b,p,m,l,w}	*na*	Ordinary place where subtransaction tokens arrive when visiting the database server. From here tokens are evenly distributed over the two database server CPUs.
B_j	{b,p,m,l,w}	$G/M/1/PS$	Queueing places used to model the two CPUs of the database server.
H	{b,p,m,l,w}	$G/M/1/FCFS$	Queueing place used to model the disk subsystem (made up of a single 100 GB disk drive) of the database server.
P	{w}	$G/M/\infty/IS$	Queueing place used to model the virtual production line stations that work orders move along during their processing. The time tokens spend here corresponds to the average delay at a production line station (i.e. 333 ms) emulated by the manufacturing application during work order processing.

subtransaction. The flow of control during the processing of subtransactions in the system is modeled by moving their respective subtransaction tokens across the different places of the QPN. Starting at place G, a dealer subtransaction token ("d") is first sent to place L, where it receives service at the CPU of the load balancer. After that, it is moved to place E and, from there, it is routed to one of the N application server CPUs represented by places A_1 to A_N. Transitions $t_{11}, t_{13}, \ldots, t_{10+N}$ have equal firing probabilities (weights) so that subtransactions are probabilistically load-balanced across the N application servers. This approximates the round-robin mechanism used by the load-balancer to distribute incoming requests among the servers. Having completed its service at the application server CPU, the dealer subtransaction token is moved to place F from where it is sent to one of the two database server CPUs with equal probability (transitions t_4 and t_5 have equal firing weights). After completing its service at the CPU, the dealer subtransaction token is moved to place H, where it receives service from the database disk subsystem. Once this is completed, the dealer subtransaction token is destroyed by transition t_8 and there are two possible scenarios:

1. A new "d" token is created in place G, which starts the next dealer subtransaction.
2. If there are no more subtransactions to be executed, the "D" token removed from place C_1 in the beginning of the transaction is returned. If the completed transaction is of type Purchase and it has generated a large order, additionally, a token "l" is created in place E.

Note that, since LargeOrder transactions are assumed to be executed with a single subtransaction, to simplify the model, we create the subtransaction token ("l") directly instead of first creating a transaction token ("L"). So, in practice, "L" tokens are not used explicitly in the model. After a "D" token of a completed transaction returns back to place C_1, it spends some time at the IS queue of the latter. This corresponds to the dealer think time. Once the dealer think time has elapsed, the "D" token is moved to the depository and the next transaction is started.

When a WorkOrder transaction is started on a planned line in the manufacturing domain, transition t_0 is fired destroying a "W" token from the depository of place C_2 and creating a "w" token in place E, which corresponds to starting the first subtransaction. Since WorkOrder subtransaction requests are load-balanced transparently (by the EJB client stubs) without using a load balancer, the WorkOrder subtransaction token ("w") is routed directly to the application server CPUs—places A_1 to A_N. It then moves along the places representing the application server and database server resources in exactly the same way as dealer subtransaction tokens. After it completes its service at place H, the following two scenarios are possible:

1. The "w" token is sent to place P whose IS queue delays it for 333 ms, corresponding to the delay at a virtual production line station. After that, the token is destroyed by transition t_{10} and a new "w" token is created in place E, representing the next WorkOrder subtransaction.
2. If there are no more subtransactions to be executed, the "w" token is destroyed by transition t_9 and the "W" token removed from place C_2 in the beginning of the transaction is returned.

After a "W" token of a completed transaction returns back to place C_2, it spends some time at the IS queue of the latter. This corresponds to the time waited after completing a work order at a production line before starting the next one. Once this time has elapsed, the "W" token is moved to the depository and the next transaction is started.

All transitions of the model are immediate and their firing modes, except for transitions t_8 and t_9, are shown in Table 5. The symbols In and Out are used here to refer to the input and output places of transitions, respectively. We assign the same firing weight (more specifically 1) to all modes of these transitions so that they have the same probability of being fired when multiples of them are

TABLE 5
Firing Modes of Transitions $t_0, t_1, \ldots, t_7$ and $t_{10}, t_{11}, \ldots, t_{11+N}$

Transition(s)	Modes and Respective Actions
t_0	1: $In\{W\} \rightarrow Out\{w\}$
t_1	1: $In\{B\} \rightarrow Out\{b\}$ 2: $In\{P\} \rightarrow Out\{p\}$ 3: $In\{M\} \rightarrow Out\{m\}$
t_2, t_3	1: $In\{b\} \rightarrow Out\{b\}$ 2: $In\{p\} \rightarrow Out\{p\}$ 3: $In\{m\} \rightarrow Out\{m\}$
t_4, t_5, t_6, t_7; $t_{11}, t_{12}, ..., t_{11+N}$	1: $In\{b\} \rightarrow Out\{b\}$ 2: $In\{p\} \rightarrow Out\{p\}$ 3: $In\{m\} \rightarrow Out\{m\}$ 4: $In\{l\} \rightarrow Out\{l\}$ 5: $In\{w\} \rightarrow Out\{w\}$
t_{10}	1: $In\{w\} \rightarrow Out\{w\}$

enabled at the same time. The definition of the firing modes of transitions t_8 and t_9 is a little more complicated. The firing modes are described in Table 6 and Table 7, respectively. The assignment of weights to the modes of these transitions is critical to achieving the desired behavior of transactions in the model. Weights must be assigned in such a way that transactions are terminated only after all of their subtransactions have been completed. We will now explain how this is done, starting with transition t_9 since this is the simpler case. According to Section 3.5.4 (Fig. 6), WorkOrder transactions are comprised of four subtransactions. This means that, for every WorkOrder transaction, four subtransactions have to be executed before the transaction is completed. To model this behavior, the firing weights (probabilities) of modes 1 and 2 are set to $3/4$ and $1/4$, respectively. Thus, out of every four times a "w" token arrives in place H and enables transition t_9, on average, the latter will be fired three times in mode 1 and one time in mode 2, completing a WorkOrder transaction. Even though the order of these firings is not guaranteed, the resulting model closely approximates the real system in terms of resource consumption and queueing behavior.

Transition t_8, on the other hand, has eight firing modes, as shown in Table 6. According to Section 3.5.4 (Fig. 5 and Fig. 6), Browse transactions have 17 subtransactions, whereas Purchase and Manage have only five. This means that, for every Browse transaction, 17 subtransactions have to be executed before the transaction is completed, i.e., out of every 17 times a "b" token arrives in place H and enables transition t_8, the latter has to be fired 16 times in mode 1 and one time in mode 2, completing a Browse transaction. Similarly, out of every five times an "m" token arrives in place H and enables transition t_8, the latter has to be fired four times in mode 6 and one time in mode 7, completing a Manage transaction. Out of every five times a "p" token arrives in place H and enables transition t_8, the latter has to be fired four times in mode 3 and one time in mode 4 or mode 5, depending on whether a large order has been generated. On average, 10 percent of all completed Purchase transactions generate large orders. Modeling these conditions probabilistically leads to a system of simultaneous equations that the firing weights (probabilities) of transition t_8 need to fulfill. One possible solution is the following: $w(1) = 16$, $w(2) = 1$, $w(3) = 13.6$, $w(4) = 3.06$, $w(5) = 0.34$, $w(6) = 13.6$, $w(7) = 3.4$, $w(8) = 17$.

The workload intensity and service demand parameters from Section 3.5.5 (Table 2 and Table 3) are used to provide values for the service times of tokens at the various queues of the model. A separate set of parameter values is specified for each workload scenario considered. The service times of subtransactions at the queues of the model are estimated by dividing the total service demands of the respective transactions by the number of subtransactions they have.

3.7 Validate, Refine, and/or Calibrate the Model

The model developed in the previous sections is now validated by comparing its predictions against measurements on the real system. Two application server nodes are available for the validation experiments. The model predictions are verified for a number of different scenarios under different transaction mixes and workload intensities. The model input parameters for two specific scenarios considered here are shown in Table 8.

The model was analyzed by means of simulation using SimQPN—our QPN analysis tool. The method of

TABLE 6
Firing Modes of Transition t_8

Mode	Action	Case Modeled
1	$H\{b\} \rightarrow G\{b\}$	Browse subtransaction has been completed. Parent transaction is not finished yet.
2	$H\{b\} \rightarrow C_1\{B\}$	Browse subtransaction has been completed. Parent transaction is finished.
3	$H\{p\} \rightarrow G\{p\}$	Purchase subtransaction has been completed. Parent transaction is not finished yet.
4	$H\{p\} \rightarrow C_1\{P\}$	Purchase subtransaction has been completed. Parent transaction is finished.
5	$H\{p\} \rightarrow C_1\{P\} + E\{l\}$	Same as (4), but assuming that completed transaction has generated a large order.
6	$H\{m\} \rightarrow G\{m\}$	Manage subtransaction has been completed. Parent transaction is not finished yet.
7	$H\{m\} \rightarrow C_1\{M\}$	Manage subtransaction has been completed. Parent transaction is finished.
8	$H\{l\} \rightarrow \{\}$	LargeOrder transaction has been completed. Its token is simply destroyed.

TABLE 7
Firing Modes of Transition t_9

Mode	Action	Case Modeled
1	$H\{w\} \rightarrow P\{w\}$	WorkOrder subtransaction has been completed. Parent transaction is not finished yet.
2	$H\{w\} \rightarrow C_2\{W\}$	WorkOrder subtransaction has been completed. Parent transaction is finished.

TABLE 8
Input Parameters for Validation Scenarios

Parameter	Scenario 1	Scenario 2
Browse Clients	20	40
Purchase Clients	10	20
Manage Clients	10	30
Planned Lines	30	50
Dealer Think Time	5 sec	5 sec
Mfg Think Time	10 sec	10 sec

nonoverlapping batch means was used for steady state analysis. Both the variation of point estimates from multiple runs of the simulation and the variation of measured performance metrics from multiple tests were negligible. For all metrics, the standard deviation of estimates was less than 2 percent of the respective mean value. Table 9 compares the model predictions against measurements on the real system. The metrics considered are transaction throughput (X_i), transaction response time (R_i), and server utilization (U_{LB} for the load balancer, U_{AS} for the application servers, and U_{DB} for the database server). The maximum modeling error for throughput is 9.3 percent, for utilization, 9.1 percent, and, for response time, 12.9 percent. Varying the transaction mix and workload intensity led to predictions of similar accuracy. Since these results are reasonable, the model is considered valid. However, even though the model is deemed valid at this point of the study, as we will see later, the model might lose its validity when it is modified in order to reflect changes in the system.

3.8 Use Model to Predict System Performance

In Section 3.3, some concrete goals were set for the performance study. The system model is now used to predict the performance of the system for the deployment configurations and workload scenarios of interest. In order to validate our approach, for each scenario considered, we will compare the model predictions against measurements on the real system. Note that this validation is not part of the methodology itself and normally it does not have to be done. Indeed, if we would have to validate the model results for every scenario considered, there would be no point in using the model in the first place. The reason we validate the model results here is to demonstrate the effectiveness of our modeling approach and showcase the predictive power of the QPN models it is based on.

As in the validation experiments, for all scenarios considered in this section, the model is analyzed by means of simulation using SimQPN and the method of nonoverlapping batch means is used for steady state analysis. Both the variation of point estimates from multiple runs of the simulation and the variation of measured performance metrics from multiple tests are negligible. For all metrics, the standard deviation of estimates is less than 2 percent of the respective mean value. Table 10 reports the analysis results for the scenarios under normal operating conditions

TABLE 9
Validation Results

	Validation Scenario 1			Validation Scenario 2		
METRIC	Model (99% c.i.)	Measured	Est. Error	Model (99% c.i.)	Measured	Est. Error
X_B	3.787 (+/- 0.042)	3.718	+1.9%	7.032 (+/- 0.141)	6.913	+1.7%
X_P	1.942 (+/- 0.021)	1.963	-1.1%	3.759 (+/- 0.075)	3.808	-1.3%
X_M	1.959 (+/- 0.022)	1.988	-1.5%	5.633 (+/- 0.113)	5.530	+1.9%
X_W	2.704 (+/- 0.030)	2.680	+0.9%	4.478 (+/- 0.090)	4.510	-0.7%
X_L	0.194 (+/- 0.002)	0.214	-9.3%	0.381 (+/- 0.008)	0.383	-0.5%
R_B	289ms (+/- 1.7ms)	256ms	+12.9%	711ms (+/- 9.5ms)	660ms	+7.7%
R_P	130ms (+/- 1.1ms)	120ms	+8.3%	311ms (+/- 5.4ms)	305ms	+2.0%
R_M	139ms (+/- 1.2ms)	130ms	+6.9%	333ms (+/- 5.5ms)	312ms	+6.7%
R_W	1086ms (+/- 0.7ms)	1108ms	-2.0%	1199ms (+/- 3.3ms)	1209ms	-0.8%
U_{LB}	20.0% (+/- 0.2)	19.5%	+2.6%	39.4% (+/- 0.3)	40.3%	-2.2%
U_{AS}	41.1% (+/- 0.3)	38.5%	+6.8%	82.0% (+/- 0.4)	83.0%	-1.2%
U_{DB}	9.6% (+/- 0.1)	8.8%	+9.1%	19.3% (+/- 0.3)	19.3%	0.0%

TABLE 10
Analysis Results for Scenarios under Normal Conditions with Four and Six Application Server Nodes

	4 App. Server Nodes			6 App. Server Nodes		
METRIC	Model (99% c.i.)	Measured	Est. Error	Model (99% c.i.)	Measured	Est. Error
X_B	7.567 (+/- 0.083)	7.438	+1.7%	7.605 (+/- 0.084)	7.415	+2.6%
X_P	3.123 (+/- 0.034)	3.105	+0.6%	3.123 (+/- 0.034)	3.038	+2.8%
X_M	3.119 (+/- 0.034)	3.068	+1.7%	3.118 (+/- 0.034)	2.993	+4.2%
X_W	4.523 (+/- 0.050)	4.550	-0.6%	4.521 (+/- 0.050)	4.320	+4.7%
X_L	0.308 (+/- 0.003)	0.318	-3.1%	0.311 (+/- 0.003)	0.307	+1.3%
R_B	298ms (+/- 1.4ms)	282ms	+5.7%	265ms (+/- 1.1ms)	267ms	-0.7%
R_P	132ms (+/- 1.0ms)	119ms	+10.9%	118ms (+/- 0.9ms)	110ms	+7.3%
R_M	141ms (+/- 1.1ms)	131ms	+7.6%	125ms (+/- 1.0ms)	127ms	-1.6%
R_W	1086ms (+/- 0.6ms)	1109ms	-2.1%	1077ms (+/- 0.5ms)	1100ms	-2.1%
U_{LB}	38.5% (+/- 0.3)	38.0%	+1.3%	38.7% (+/- 0.3)	38.5%	+0.5%
U_{AS}	38.0% (+/- 0.3)	35.8%	+6.1%	25.5% (+/- 0.2)	23.7%	+7.6%
U_{DB}	16.7% (+/- 0.2)	18.5%	-9.7%	16.8% (+/- 0.2)	15.5%	+8.4%

TABLE 11
Load Balancer Service Demands

Load Balancer	Browse	Purchase	Manage
Original	42.72ms	9.98ms	9.93ms
Upgraded	32.25ms	8.87ms	8.56ms

with four and six application server nodes. In both cases, the model predictions are very close to the measurements on the real system. Even for response time, the metric with highest variation, the modeling error, does not exceed 10.9 percent.

Table 12 shows the model predictions for two scenarios under peak conditions with six application server nodes. The first one uses the original load balancer, while the second one uses an upgraded load balancer with a faster CPU. The faster CPU results in lower service demands, as shown in Table 11. With the original load balancer, six application server nodes turned out to be insufficient to guarantee average response times of business transactions below half a second. However, with the upgraded load balancer, this was achieved. In the rest of the scenarios considered, the upgraded load balancer will be used.

We now consider the behavior of the system as the workload intensity increases beyond peak conditions and further application server nodes are added. Table 13 shows the model predictions for two scenarios with an increased number of concurrent Browse clients, i.e., 150 in the first one and 200 in the second one. In both scenarios, the number of application server nodes is eight. As evident from the results, the load balancer is completely saturated when increasing the workload intensity and it becomes a bottleneck, limiting the overall system performance. Therefore, adding further application server nodes would not bring any benefit, unless the load balancer is replaced with a faster one.

3.9 Modeling Thread Contention

Since the load balancer is the bottleneck resource, it is interesting to investigate its behavior a little further. Until now, it was assumed that, when a request arrives at the load balancer, there is always a free thread which can start processing it immediately. However, if one keeps increasing the workload intensity, the number of concurrent requests at the load balancer will eventually exceed the number of available threads. The latter would lead to thread contention, resulting in additional delays at the load balancer, not captured by our system model. This is a typical example of how a valid model may lose its validity as the workload evolves. We will now show how the model can be refined to capture the thread contention at the load balancer.

TABLE 12
Analysis Results for Scenarios under Peak Conditions with Six Application Server Nodes

	Original Load Balancer			Upgraded Load Balancer		
METRIC	Model (99% c.i.)	Measured	Est. Error	Model (99% c.i.)	Measured	Est. Error
X_B	17.879 (+/- 0.805)	17.742	+0.8%	18.444 (+/- 0.646)	18.347	+0.5%
X_P	5.005 (+/- 0.225)	4.913	+1.9%	4.995 (+/- 0.175)	5.072	-1.5%
X_M	5.006 (+/- 0.227)	4.995	-0.2%	5.035 (+/- 0.176)	5.032	+0.1%
X_W	9.034 (+/- 0.407)	8.880	+1.7%	9.002 (+/- 0.315)	8.850	+1.7%
X_L	0.519 (+/- 0.023)	0.490	+5.9%	0.505 (+/- 0.018)	0.515	-1.9%
R_B	568ms (+/- 14.1ms)	534ms	+6.4%	418ms (+/- 6.9ms)	440ms	-5.0%
R_P	213ms (+/- 6.2ms)	198ms	+7.6%	182ms (+/- 4.6ms)	165ms	+10.3%
R_M	227ms (+/- 6.5ms)	214ms	+6.1%	196ms (+/- 4.6ms)	187ms	+4.8%
R_W	1112ms (+/- 2.3ms)	1135ms	-2.0%	1115ms (+/- 2.4ms)	1123ms	-0.7%
U_{LB}	86.7% (+/- 0.4)	88.0%	-1.5%	68.6% (+/- 0.3)	70.0%	-2.0%
U_{AS}	54.2% (+/- 0.2)	53.8%	+0.7%	55.7% (+/- 0.2)	55.3%	+0.7%
U_{DB}	33.0% (+/- 0.2)	34.5%	-4.3%	33.6% (+/- 0.2)	35.0%	-4.0%

TABLE 13
Analysis Results for Scenarios under Heavy Load with Eight Application Server Nodes

	Heavy Load Scenario 1			Heavy Load Scenario 2		
METRIC	Model (99% c.i.)	Measured	Est. Error	Model (99% c.i.)	Measured	Est. Error
X_B	26.419 (+/- 1.189)	25.905	+2.0%	28.414 (+/- 0.994)	26.987	+5.3%
X_P	4.936 (+/- 0.222)	4.817	+2.5%	4.606 (+/- 0.161)	4.333	+6.3%
X_M	4.935 (+/- 0.220)	4.825	+2.3%	4.610 (+/- 0.162)	4.528	+1.8%
X_W	8.989 (+/- 0.405)	8.820	+1.9%	8.963 (+/- 0.314)	8.970	+0.1%
X_L	0.508 (+/- 0.023)	0.488	+4.1%	0.455 (+/- 0.016)	0.417	+9.1%
R_B	655ms (+/- 14.5ms)	714ms	-8.3%	2043ms (+/- 31.5ms)	2288ms	-10.7%
R_P	250ms (+/- 7.6ms)	257ms	-2.7%	637ms (+/- 14.3ms)	802ms	-20.6%
R_M	259ms (+/- 7.9ms)	276ms	-6.2%	633ms (+/- 14.2ms)	745ms	-15.0%
R_W	1116ms (+/- 2.1ms)	1128ms	-1.1%	1123ms (+/- 2.3ms)	1132ms	-0.8%
U_{LB}	93.8% (+/- 0.2)	95.0%	-1.3%	99.9% (+/- 0.1)	100.0%	0.0%
U_{AS}	54.2% (+/- 0.4)	54.1%	+0.2%	57.3% (+/- 0.3)	55.7%	+2.9%
U_{DB}	38.8% (+/- 0.2)	42.0%	-7.6%	39.6% (+/- 0.2)	42.0%	-5.7%

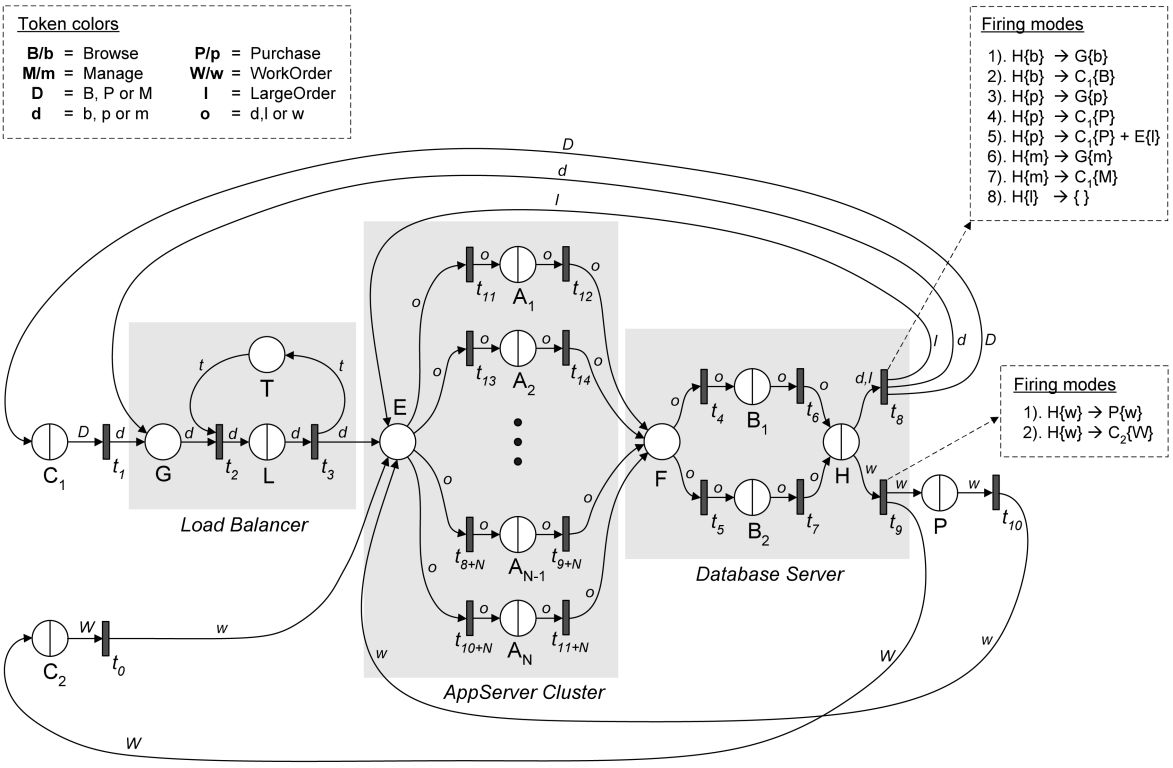


Fig. 9. Extended QPN model of the system (capturing thread contention at the load balancer).

3.9.1 Extending the System Model

As discussed in Section 2.2, passive system resources such as threads can be modeled as tokens inside ordinary places. In Fig. 9, an extended version of our system model is shown which includes an ordinary place, T, representing the load balancer thread pool. Before a dealer request is scheduled for processing at the load balancer CPU, a token "t" representing a thread is allocated from the thread pool. After the dealer request has been served at the load balancer CPU, the token is returned back to the pool. Thus, if an arriving request finds no available thread, it will have to wait in place G until a thread is released. The initial population of place T determines the number of threads in the thread pool.

At first sight, this appears to be the right approach to model the thread contention at the load balancer. However, an attempt to validate the extended model reveals a significant discrepancy between the model predictions and measurements on the real system. In particular, it stands out that predicted response times are much lower than measured response times for dealer transactions with low workload intensities. A closer investigation shows that the problem is in the way dealer subtransaction tokens arriving in place G are scheduled for processing at the load balancer CPU. Dealer subtransaction tokens become available for firing of transition t_2 immediately upon their arrival at place G. Thus, whenever arriving tokens are blocked in place G, their order of arrival is lost. After a thread is released, transition t_2 fires in one of its enabled modes with equal probability. Therefore, the order in which waiting subtransaction tokens are scheduled for processing does not match the order of their arrival at place G. This obviously does not reflect the way the real system works and renders the model unrepresentative.

3.9.2 Introducing QPN Departure Disciplines

The above situation describes a common drawback of Petri net models, i.e., tokens inside ordinary places are not distinguished in terms of their order of arrival. One approach to address the problem would be to replace the ordinary place G with an immediate queueing place containing an FCFS queue. However, simply using an FCFS queue would not resolve the problem since arriving tokens would be served immediately and moved to the depository, where their order of arrival will still be lost. To address this, we could exploit the generalized queue definition in [22] to define the scheduling strategy of place G's queue in such a way that tokens are served immediately according to FCFS, but only if the depository is empty. If there is a token in the depository, all tokens are blocked in their current position until the depository becomes free. Even though this would theoretically address the issue with the token order, it would create another problem. The available tools and techniques for QPN analysis, including SimQPN, do not support queues with scheduling strategy dependent on the state of the depository. Indeed, the generalized queue definition given in [22], while theoretically powerful, is impractical to implement, so, in practice, it is rarely used and queues in QPNs are usually treated as conventional queues from queueing network theory. The way we address the problem is by introducing *departure disciplines*, which are a simple yet powerful feature we have added to SimQPN. The departure discipline of an ordinary place or depository determines the order in which arriving tokens become

TABLE 14
Workload Intensity Parameters for
Heavy Load Scenarios with Thread Contention

Parameter	Heavy Load Sc. 3	Heavy Load Sc. 4
Browse Clients	300	270
Purchase Clients	30	90
Manage Clients	30	60
Planned Lines	120	120
Dealer Think Time	5 sec	5 sec
Mfg Think Time	10 sec	10 sec

available for output transitions. We define two departure disciplines, Normal (used by default) and First-In-First-Out (FIFO). The former implies that tokens become available for output transitions immediately upon arrival, just like in conventional QPN models. The latter implies that tokens become available for output transitions in the order of their arrival, i.e., a token can leave the place/depository only after all tokens that have arrived before it have left, hence the term FIFO.

Coming back to the problem above with the way thread contention is modeled, we now change place G to use the FIFO departure discipline. This ensures that subtransaction tokens waiting at place G are scheduled for processing in the order in which they arrive. After this change, the model passes the validation tests and can be used for performance prediction.

3.9.3 Performance Prediction

We consider two additional heavy load scenarios with an increased number of concurrent dealer clients leading to thread contention in the load balancer. The workload intensity parameters for the two scenarios are shown in Table 14.

The first scenario has a total of 360 concurrent dealer clients, the second 420. Table 15 compares the model predictions for the first scenario in two configurations with eight application servers and 15 and 30 load balancer threads, respectively. In addition to response times, throughput, and utilization, the average length of the load balancer thread queue (N_{LBTQ}) is considered. As evident from the results, the model predictions are very close to the measurements and, even for response times, the modeling error does not exceed 16.4 percent. The results for the second scenario look very similar. The CPU utilization of the WebLogic servers and the database server increase to 63 percent and 52 percent, respectively, leading to slightly higher response times and lower throughput. The modeling error does not exceed 15.2 percent. For lack of space, we do not include the detailed results. Repeating the analysis for a number of variations of the model input parameters led to results of similar accuracy.

3.10 Analyze Results and Address Modeling Objectives

We can now use the results from the performance analysis to address the goals established in Section 3.3. By means of the developed QPN model, we were able to predict the performance of the system under normal operating conditions with four and six WebLogic servers. It turned out that, using the original load balancer, six WebLogic servers were insufficient to guarantee average response times of business transactions below half a second. Upgrading the load balancer with a slightly faster CPU led to the CPU utilization of the load balancer dropping by a good 20 percent. As a result, the response times of dealer transactions improved by 14 to 26 percent, meeting the "half a second" requirement. However, increasing the workload intensity beyond peak conditions revealed that the load balancer was a bottleneck resource, preventing us from scaling the system by adding additional WebLogic servers (see Fig. 10). Therefore, in light of the expected workload growth, the company should either replace the load balancer machine with a faster one or consider using a more efficient load balancing method. After this is done, the performance analysis should be repeated with the new load balancer to make sure that there are no other system bottlenecks. It should also be ensured that the load balancer is configured with enough threads to prevent thread contention.

TABLE 15
Analysis Results for Heavy Load Scenario 3 with 15 and 30 Load Balancer Threads and Eight Application Server Nodes

	Heavy Load Sc. 3 with 15 Thr.			Heavy Load Sc. 3 with 30 Thr.		
METRIC	Model (99% c.i.)	Measured	Est. Error	Model (99% c.i.)	Measured	Est. Error
X_B	28.602 (+/- 1.001)	27.323	+4.7%	28.576 (+/- 1.000)	27.205	+5.0%
X_P	4.480 (+/- 0.157)	4.220	+6.2%	4.487 (+/- 0.157)	4.213	+6.5%
X_M	4.497 (+/- 0.159)	4.387	+2.5%	4.528 (+/- 0.158)	4.485	+1.0%
X_W	10.771 (+/- 0.377)	10.660	+1.0%	10.810 (+/- 0.378)	10.800	+0.1%
X_L	0.448 (+/- 0.016)	0.410	+9.3%	0.440 (+/- 0.015)	0.446	+0.1%
R_B	5494ms (+/- 40.8ms)	5740ms	-4.3%	5519ms (+/- 39.5ms)	5805ms	-4.9%
R_P	1673ms (+/- 16.3ms)	1977ms	-15.4%	1672ms (+/- 16.0ms)	2001ms	-16.4%
R_M	1683ms (+/- 17.3ms)	1779ms	-5.4%	1676ms (+/- 17.2ms)	1801ms	-6.9%
R_W	1124ms (+/- 2.2ms)	1158ms	-2.9%	1124ms (+/- 2.3ms)	1143ms	-1.7%
U_{LB}	99.9% (+/- 0.1)	93.0%	+7.5%	99.9% (+/- 0.1)	100.0%	0.0%
U_{AS}	57.8% (+/- 0.3)	57.8%	0.0%	57.8% (+/- 0.3)	58.0%	-0.3%
U_{DB}	41.5% (+/- 0.2)	44.0%	-5.7%	41.6% (+/- 0.2)	44.0%	-5.5%
N_{LBTQ}	146 (+/- 5.1)	161	-9.3%	131 (+/- 4.6)	146	-10.3%

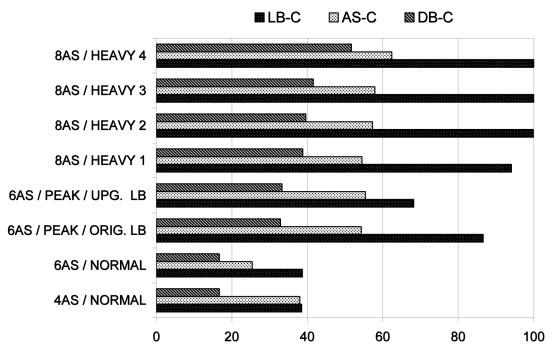


Fig. 10. Predicted server CPU utilization in considered scenarios.

4 Summary

In this paper, we presented a novel case study of a realistic DCS, showing how enhanced QPN models can be exploited as a powerful performance prediction tool in the software engineering process. Along with the case study, we presented a practical performance modeling methodology which helps to construct models that accurately reflect the system performance and scalability characteristics. The paper started with an overview of the methodology which is based on existing work in software performance engineering. We described the main steps that are normally followed, concentrating on the aspects which are specific to our approach. After that, we presented the case study which is the main contribution of the paper. A deployment of the industry-standard SPECjAppServer2004 benchmark was studied, a large and complex application designed to be representative of today's real-world DCS. It was shown in a step-by-step fashion how to build a detailed QPN model of the system, validate it, and then use it to evaluate the system performance and scalability. In addition to CPU and I/O contention, it was demonstrated how some complex aspects of system behavior, such as composite transactions, software contention, and asynchronous processing, can be modeled. The developed QPN model was analyzed for a number of different deployment configurations and workload scenarios. The models demonstrated much better scalability and predictive power than what was achieved in our previous work. Even for the largest and most complex scenarios, the modeling error for transaction response time did not exceed 20.6 percent and was much lower for transaction throughput and resource utilization.

Appendix

Introduction to Queuing Petri Nets

Queueing Petri nets can be seen as a combination of a number of different extensions to conventional Petri nets (PNs) along several different dimensions. In this section, we include some basic definitions and briefly discuss how queueing Petri nets have evolved. A more detailed treatment of the subject can be found in [18], [22]. An ordinary *Petri net* is a bipartite directed graph composed of places, drawn as circles, and transitions, drawn as bars. A formal definition follows [18]:

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

1. $P = \{p_1, p_2, \ldots, p_n\}$ *is a finite and nonempty set of* places,
2. $T = \{t_1, t_2, \ldots, t_m\}$ *is a finite and nonempty set of* transitions, $P \cap T = \emptyset$,
3. $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$ *are called backward and forward* incidence functions, *respectively*,
4. $M_0 : P \rightarrow \mathbb{N}_0$ *is called* initial marking.

Different extensions to ordinary PNs have been developed in order to increase the modeling convenience and/or the modeling power. *Colored PNs (CPNs)*, introduced by Jensen [31], 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, specifying the types of tokens that can reside in the place. 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 modes to each transition and incidence functions are defined on a per mode basis. A formal definition of a CPN follows [18]:

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

1. $P = \{p_1, p_2, \ldots, p_n\}$ *is a finite and nonempty set of* places,
2. $T = \{t_1, t_2, \ldots, t_m\}$ *is a finite and nonempty set of* transitions, $P \cap T = \emptyset$,
3. C *is a* color function *that assigns a finite and nonempty set of* colors *to each place and a finite and nonempty set of* modes *to each transition*,
4. I^- *and* I^+ *are the backward and forward* incidence functions *defined on* $P \times T$ *such that*

$$I^-(p,t), I^+(p,t) \in [C(t) \rightarrow C(p)_{MS}],$$

$\forall (p,t) \in P \times T$,[5]

5. M_0 *is a function defined on* P *describing the* initial marking *such that* $M_0(p) \in C(p)_{MS}$.

Other extensions to ordinary PNs allow temporal (timing) aspects to be integrated into the net description [18]. In particular, *Stochastic PNs (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 PNs (GSPNs)* allow two types of transitions to be used: immediate and timed. Once enabled, immediate transitions fire in zero time. If several immediate transitions 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 a GSPN follows [18]:

Definition 3. *A Generalized SPN (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 PN*,
2. $T_1 \subseteq T$ *is the set of timed transitions*, $T_1 \neq \emptyset$,

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

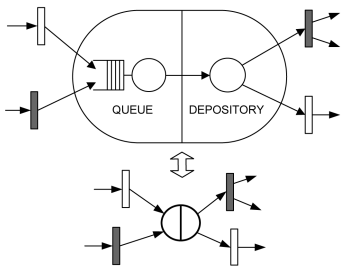


Fig. 11. A queueing place and its shorthand notation.

3. $T_2 \subset T$ *is the set of immediate transitions,* $T_1 \cap T_2 = \emptyset$, $T_1 \cup T_2 = T$,
4. $W = (w_1, \ldots, w_{|T|})$ *is an array whose entry* $w_i \in \mathbb{R}^+$ *is a rate of a negative exponential distribution specifying the firing delay, if* $t_i \in T_1$ *or is a firing weight specifying the relative firing frequency, if* $t_i \in T_2$.

Combining Definitions 2 and 3 leads to *Colored GSPNs (CGSPNs)* [18]:

Definition 4. *A Colored GSPN (CGSPN) is a 4-tuple* $CGSPN = (CPN, T_1, T_2, W)$, *where:*

1. $CPN = (P, T, C, I^-, I^+, M_0)$ *is the underlying CPN,*
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,* $T_1 \cap T_2 = \emptyset$, $T_1 \cup T_2 = T$,
4. $W = (w_1, \ldots, w_{|T|})$ *is an array with* $w_i \in [C(t_i) \longmapsto \mathbb{R}^+]$ *such that* $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$ *is a rate of a negative exponential distribution specifying the firing delay due to color c, if* $t_i \in T_1$ *or is a firing weight specifying the relative firing frequency due to c, if* $t_i \in T_2$.

While CGSPNs have proven to be a very powerful modeling formalism, they do not provide any means for direct representation of queueing disciplines. The attempts to eliminate this disadvantage have led to the emergence of *Queueing PNs (QPNs)*. The main idea behind the QPN modeling paradigm was to add queueing and timing aspects to the places of CGSPNs. This is done by allowing queues (service stations) to be integrated into places of CGSPNs. A place of a CGSPN that 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 Fig. 11.

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. The rest of the net behaves like a normal CGSPN. A formal definition of a QPN follows:

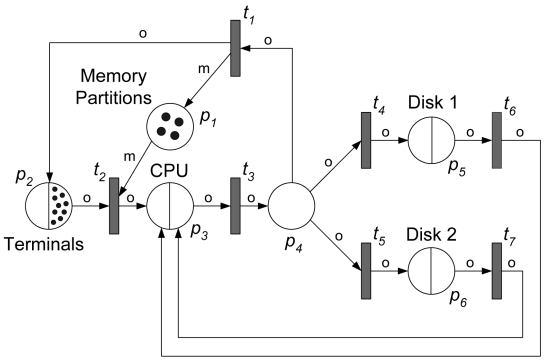


Fig. 12. A QPN model of a central server with memory constraints (reprinted from [18]).

Definition 5. *A Queueing PN (QPN) is an 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 PN,*
2. $Q = (\tilde{Q}_1, \tilde{Q}_2, (q_1, \ldots, q_{|P|}))$, *where*
   - $\tilde{Q}_1 \subseteq P$ *is the set of timed queueing places,*
   - $\tilde{Q}_2 \subseteq P$ *is the set of immediate queueing places,* $\tilde{Q}_1 \cap \tilde{Q}_2 = \emptyset$, *and*
   - q_i *denotes the description of a queue*[6] *taking all colors of* $C(p_i)$ *into consideration, if* p_i *is a queueing place or equals the keyword "null" if* p_i *is an ordinary place.*
3. $W = (\tilde{W}_1, \tilde{W}_2, (w_1, \ldots, w_{|T|}))$, *where*
   - $\tilde{W}_1 \subseteq T$ *is the set of timed transitions,*
   - $\tilde{W}_2 \subseteq T$ *is the set of immediate transitions,* $\tilde{W}_1 \cap \tilde{W}_2 = \emptyset$, $\tilde{W}_1 \cup \tilde{W}_2 = T$, *and*
   - $w_i \in [C(t_i) \longmapsto \mathbb{R}^+]$ *such that* $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$ *is interpreted as a rate of a negative exponential distribution specifying the firing delay due to color c, if* $t_i \in \tilde{W}_1$ *or a firing weight specifying the relative firing frequency due to color c, if* $t_i \in \tilde{W}_2$.

Example 1 (QPN). Fig. 12 shows an example of a QPN model of a central server system with memory constraints based on [18]. Place p_2 represents several terminals where users start jobs (modeled with tokens of color "o") after a certain thinking time. These jobs request service at the CPU (represented by a G/C/1/PS queue, where C stands for Coxian distribution) and two disk subsystems (represented by G/C/1/FCFS queues). To enter the system, each job has to allocate a certain amount of memory. 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.

6. In the most general definition of QPNs, queues are defined in a very generic way, allowing the specification of arbitrarily complex scheduling strategies taking into account the state of both the queue and the depository of the queueing place [22]. For the purposes of this paper, it is enough to use conventional queues as defined in queueing network theory.

According to Definition 5, we have the following: $QPN = (P, T, C, I^-, I^+, M_0, Q, W)$, where

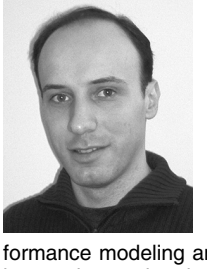
- $CPN = (P, T, C, I^-, I^+, M_0)$ is the underlying Colored PN, as depicted in Fig. 12,
- $$Q = (\tilde{Q}_1, \tilde{Q}_2, (null, \mathrm{G/C/}\infty\mathrm{/IS}, \mathrm{G/C/1/PS}, null, \mathrm{G/C/1/FCFS}, \mathrm{G/C/1/FCFS})),$$
  $\tilde{Q}_1 = \{p_2, p_3, p_5, p_6\}, \tilde{Q}_2 = \emptyset$,
- $W = (\tilde{W}_1, \tilde{W}_2, (w_1, \ldots, w_{|T|}))$, where $\tilde{W}_1 = \emptyset$, $\tilde{W}_2 = T$, and $\forall c \in C(t_i) : w_i(c) := 1$ so that all transition firings are equally likely.

ACKNOWLEDGMENTS

The author gratefully acknowledges the many fruitful discussions with Dr. Falko Bause from the University of Dortmund. He would also like to thank the anonymous reviewers for their thoughtful comments and suggestions which helped to improve the quality of the paper. This work was partially funded by BEA Systems, Inc. as a part of the project "Performance Modeling and Evaluation of Large-Scale J2EE Applications" and the German Research Foundation (DFG) as part of the PhD program "Enabling Technologies for E-Commerce" at Darmstadt University of Technology.

REFERENCES

[1] C. Smith, "Performance Engineering," *Encyclopedia of Software Eng.*, J.J. Maciniak, ed., pp. 794-810, John Wiley & Sons, 1994.
[2] D. Menascé, V. Almeida, and L. Dowdy, *Performance by Design.* Prentice Hall, 2004.
[3] S. Kounev and A. Buchmann, "Performance Modeling and Evaluation of Large-Scale J2EE Applications," *Proc. 29th Int'l Computer Measurement Groupt (CMG) Conf.*, 2003.
[4] S. Kounev and A. Buchmann, "Performance Modelling of Distributed E-Business Applications Using Queueing Petri Nets," *Proc. 2003 IEEE Int'l Symp. Performance Analysis of Systems and Software*, 2003.
[5] S. Kounev and A. Buchmann, "SimQPN—A Tool and Methodology for Analyzing Queueing Petri Net Models by Means of Simulation," *Performance Evaluation*, vol. 63, nos. 4-5, pp. 364-394, May 2006, doi:10.1016/j.peva.2005.03.004.
[6] D. Menascé, V. Almeida, and L. Dowdy, *Capacity Planning and Performance Modeling—From Mainframes to Client-Server Systems.* Englewood Cliffs, N.J.: Prentice Hall, 1994.
[7] D. Menascé, V. Almeida, R. Fonseca, and M. Mendes, "A Methodology for Workload Characterization of E-Commerce Sites," *Proc. First ACM Conf. Electronic Commerce*, pp. 119-128, Nov. 1999.
[8] D. Menascé and V. Almeida, *Capacity Planning for Web Performance: Metrics, Models and Methods.* Upper Saddle River, N.J.: Prentice Hall, 1998.
[9] D. Menascé and V. Almeida, *Scaling for E-Business—Technologies, Models, Performance and Capacity Planning.* Upper Saddle River, N.J.: Prentice Hall, 2000.
[10] C. Smith and L. Williams, *Performance Solutions—A Practical Guide to Creating Responsive, Scalable Software.* Addison-Wesley, 2002.
[11] L. Williams and C. Smith, "PASA: An Architectural Approach to Fixing Software Problems," *Proc. 28th Computer Measurement Group (CMG) Conf.*, 2002, http://www.perfeng.com/pasa.htm.
[12] L. Williams and C. Smith, "Performance and Scalability of Distributed Software Architectures: An SPE Approach," *Parallel and Distributed Computing Practices*, 2002.
[13] M. Woodside, *Tutorial Introduction to Layered Modeling of Software Performance*, third ed., May 2002, http://www.sce.carleton.ca/rads/lqn/lqn-documentation/tutorialf.pdf.
[14] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy, "Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queueing Network Templates," *Proc. Workshop Specification and Verification of Component-Based Systems (at ESEC/FSE 05)*, 2005.
[15] P. Maly and C. Woodside, "Layered Modeling of Hardware and Software, with Application to a LAN Extension Router," *Proc. 11th Int'l Conf. Computer Performance Evaluation Techniques and Tools (TOOLS)*, 2000.
[16] C. Woodside, C. Hrischuk, B. Selic, and S. Bayarov, "Automated Performance Modeling of Software Generated by a Design Environment," *Performance Evaluation*, vol. 45, pp. 107-123, 2001.
[17] M. Woodside, J. Neilson, D. Petriu, and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software," *IEEE Trans. Computers*, vol. 44, no. 1, pp. 20-34, Jan. 1995.
[18] F. Bause and F. Kritzinger, *Stochastic Petri Nets—An Introduction to the Theory*, second ed. Vieweg Verlag, 2002.
[19] F. Bause, "'QN + PN = QPN'—Combining Queueing Networks and Petri Nets," Technical Report No. 461, Dept. of Computer Science, Univ. of Dortmund, Germany, 1993.
[20] S. Kounev, "SPECjAppServer2004—The New Way to Evaluate J2EE Performance," DEV2DEV Article, O'Reilly Publishing Group, 2005, http://dev2dev.bea.com/pub/a/2005/03/SPECjAppServer2004.html.
[21] L. Su, K. Chow, K. Shiv, and A. Jha, "A Comparison of SPECjAppServer2002 and SPECjAppServer2004," *Proc. Eighth Workshop Computer Architecture Evaluation Using Commercial Workloads (CAECW-8)*, 2005.
[22] F. Bause, "Queueing Petri Nets—A Formalism for the Combined Qualitative and Quantitative Analysis of Systems," *Proc. Fifth Int'l Workshop Petri Nets and Performance Models*, 1993.
[23] F. Bause, P. Buchholz, and P. Kemper, "Integrating Software and Hardware Performance Models Using Hierarchical Queueing Petri Nets," *Proc. Ninth ITG/GI—Fachtagung Messung, Modellierung und Bewertung von Rechen—und Kommunikationssystemen, (MMB '97)*, 1997.
[24] M. Calzarossa and G. Serazzi, "Workload Characterization: A Survey," *Proc. IEEE*, vol. 81, no. 8, pp. 1136-1150, 1993.
[25] J. Mohr and S. Penansky, "A Forecasting Oriented Workload Characterization Methodology," *CMG Trans.*, vol. 36, June 1982.
[26] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.
[27] C. Rose, "A Measurement Procedure for Queueing Network Models of Computer Systems," *ACM Computing Surveys*, vol. 10, no. 3, 1978.
[28] D. Menascé and H. Gomaa, "A Method for Design and Performance Modeling of Client/Server Systems," *IEEE Trans. Software Eng.*, vol. 26, no. 11, Nov. 2000.
[29] J. Flowers and L. Dowdy, "A Comparison of Calibration Techniques for Queueing Network Models," *Proc. 1989 Int'l Computer Measurement Group (CMG) Conf.*, pp. 644-655, 1989.
[30] Standard Performance Evaluation Corp. (SPEC), "SPECjAppServer2004 Documentation," SPEC, Apr. 2004, http://www.spec.org/jAppServer2004/.
[31] K. Jensen, *Coloured Petri Nets and the Invariant Method*, pp. 327-338. Math. Foundations on Computer Science, 1981.



Samuel Kounev received the MSc degree in mathematics and computer science from the University of Sofia (1999) and the PhD degree in computer science from Darmstadt University of Technology (2005). He is currently a postdoctoral research fellow at Cambridge University as well as a research associate in the Databases and Distributed Systems Group at Darmstadt University of Technology. His research interests include software performance engineering, performance modeling and evaluation of distributed systems, benchmarking, and capacity planning. He has served as the release manager of SPEC's Java Subcommittee since 2003 and has been involved in a number of industrial projects, including the SPECjAppServer and SPECjbb set of industry standard benchmarks. He was the recipient of various prizes and awards from industry and academia. He is a member of the IEEE Computer Society and the ACM.