On the Use of Queueing Petri Nets for Modeling and Performance Analysis of Distributed Systems

Samuel Kounev & Alejandro Buchmann
Technische Universität Darmstadt
Germany

1 Introduction

Predictive performance models are used increasingly throughout the phases of the software engineering lifecycle of distributed 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. The challenge stems from the limited model expressiveness on the one hand and the limited scalability of model analysis techniques on the other. This chapter presents a novel methodology for modeling and performance analysis of distributed systems [Kounev, 2006]. The methodology is based on queueing Petri nets (QPNs) which provide greater modeling power and expressiveness than conventional modeling paradigms such as queueing networks and generalized stochastic Petri nets. Using QPNs, one can integrate both hardware and software aspects of system behavior into the same model. In addition to hardware contention and scheduling strategies, QPNs make it easy to model software contention, simultaneous resource possession, synchronization, blocking and asynchronous processing. These aspects have significant impact on the performance of modern distributed systems.

To avoid the problem of state space explosion, our methodology uses discrete event simulation for model analysis. We propose an efficient and reliable method for simulation of QPNs [Kounev and Buchmann, 2006]. As a validation of our approach, we present a case study of a real-world distributed system, showing how our methodology is applied in a step-by-step fashion to evaluate the system performance and scalability. The system studied is a deployment of the industry-standard SPECjAppServer2004 benchmark. A detailed model of the system and its workload is built and used to predict the system performance for several deployment configurations and workload scenarios of interest. Taking advantage of the expressive power of QPNs, our approach makes it possible to model systems at a higher degree of accuracy providing a number of important benefits.

The rest of this chapter is organized as follows. In Section 2, we give a brief introduction to QPNs. Following this, in Section 3, we present a method for quantitative analysis of QPNs based on discrete event simulation. The latter enables us to analyze QPN models of realistic size and complexity. In Section 4, we present our performance modeling methodology for distributed systems. The methodology is introduced in a step-by-step fashion by considering a case study in which QPNs are used to model a real-life system and analyze its performance and scalability. After the case study, some concluding remarks are presented and the chapter is wrapped up in Section 5.

2 Queueing Petri Nets

Queueing Petri Nets (QPNs) 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 QPNs have evolved. A deeper and more detailed treatment of the subject can be found in [Bause, 1993].

2.1 Evolution of Queueing Petri Nets

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 is given below [Bause and Kritzinger, 2002]:

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

1. *$P = \{p_1, p_2, ..., p_n\}$ is a finite and non-empty set of* places*,*
2. *$T = \{t_1, t_2, ..., t_m\}$ is a finite and non-empty 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.

The incidence functions I^- and I^+ specify the interconnections 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.

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 K. Jensen are one such extension [Jensen, 1981]. 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 [Bause and Kritzinger, 2002]:

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

1. *$P = \{p_1, p_2, ..., p_n\}$ is a finite and non-empty set of* places*,*
2. *$T = \{t_1, t_2, ..., t_m\}$ is a finite and non-empty set of* transitions*, $P \cap T = \emptyset$,*
3. *C is a* color function *that assigns a finite and non-empty set of* colors *to each place and a finite and non-empty set of* modes *to each transition.*

4. *I^- and I^+ are the backward and forward* incidence functions *defined on $P \times T$, such that $I^-(p,t), I^+(p,t) \in [C(t) \rightarrow C(p)_{MS}], \forall (p,t) \in P \times T$*[1]

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 [Bause and Kritzinger, 2002]. 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 [Bause and Kritzinger, 2002]:

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, $T_1 \neq \emptyset$,*

3. *$T_2 \subset T$ is the set of immediate transitions, $T_1 \cap T_2 = \emptyset$, $T_1 \cup T_2 = T$,*

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

Combining CPNs and GSPNs leads to *Colored GSPNs (CGSPNs)* [Bause and Kritzinger, 2002]:

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. *$W = (w_1, ..., w_{|T|})$ 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 $t_i \in T_1$ $\underline{or}$ 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 Figure 1.

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

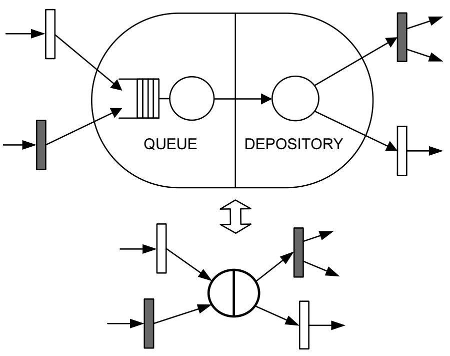

Figure 1: A queueing place and its shorthand notation.

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 *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. 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. A formal definition of a QPN follows:

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. $Q = (\tilde{Q}_1, \tilde{Q}_2, (q_1, ..., q_{|P|}))$ *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*[2] *taking all colors of* $C(p_i)$ *into consideration, if* p_i *is a queueing place <u>or</u> equals the keyword 'null', if* p_i *is an ordinary place.*
3. $W = (\tilde{W}_1, \tilde{W}_2, (w_1, ..., w_{|T|}))$ *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* $t_i \in \tilde{W}_1$ *<u>or</u> a firing weight specifying the relative firing frequency due to color c, if* $t_i \in \tilde{W}_2$.

[2] 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 [Bause, 1993]. For the purposes of this chapter, it is enough to use conventional queues as defined in queueing network theory.

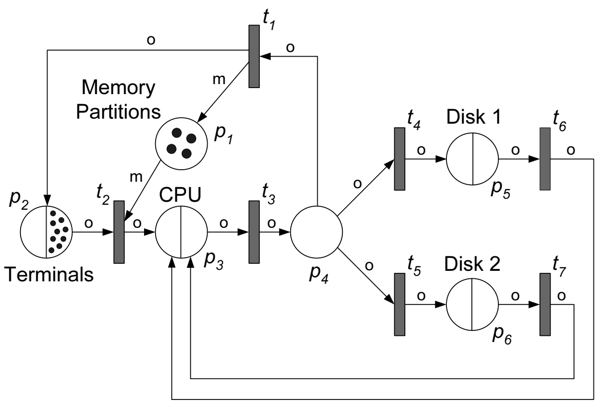

Figure 2: A QPN model of a central server with memory constraints (reprinted from [Bause and Kritzinger, 2002]).

Example 1 (QPN) *Figure 2 shows an example of a QPN model of a central server system with memory constraints based on [Bause and Kritzinger, 2002]. 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. Note that, for readability, token cardinalities have been omitted from the arc weights in Figure 2, i.e., symbol **o** stands for **1'o** and symbol **m** for **1'm**. 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 Figure 2,*
- *$Q = (\tilde{Q_1}, \tilde{Q_2}, (null, G/C/\infty/IS, G/C/1/PS, null, G/C/1/FCFS, G/C/1/FCFS))$, $\tilde{Q_1} = \{p_2, p_3, p_5, p_6\}, \tilde{Q_2} = \emptyset$,*
- *$W = (\tilde{W_1}, \tilde{W_2}, (w_1, ..., w_{|T|}))$, 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.*

2.2 Hierarchical Queueing Petri Nets

A major hurdle to the practical application of QPNs is the so-called *largeness problem* or *state-space explosion problem*: as one increases the number of queues and tokens in a QPN, the size of the model's state space 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 analytically tractable. An attempt to alleviate this problem was the introduction of *Hierarchically-Combined QPNs (HQPNs)* [Bause *et al.*, 1994]. 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 that 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 3. A subnet place might contain an ordinary QPN or again a

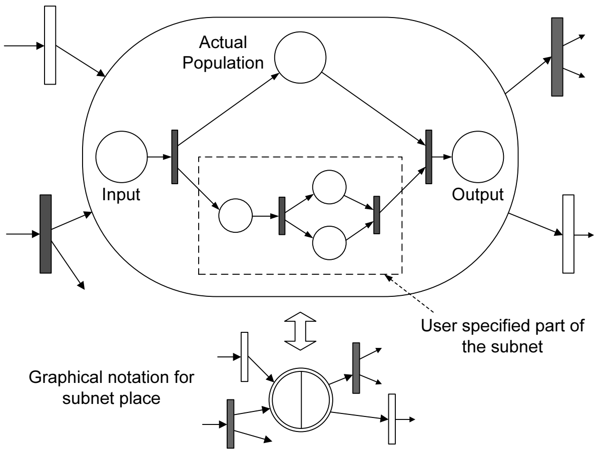
Actual
Population
Input
Output
User specified part of the subnet
Graphical notation for subnet place

Figure 3: A subnet place and its shorthand notation.

HQPN allowing multiple levels of nesting. For simplicity, 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. Every subnet of a HQPN has a dedicated input and output place, which are ordinary places of a CPN. 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 output transitions of the subnet place. Tokens contained in all other places of the HQPN subnet are not available for output transitions of the subnet place. Every HQPN subnet also contains $actual - population$ place used to keep track of the total number of tokens fired into the subnet place.

3 Quantitative Analysis of Queueing Petri Nets

In [Kounev and Buchmann, 2003], we showed that QPNs lend themselves very well to modeling distributed e-business applications with software contention and demonstrated how this can be exploited for performance prediction in the capacity planning process. However, we also showed that modeling a realistic e-business application using QPNs often leads to a model that is way too large to be analytically tractable. While, HQPNs and structured analysis techniques alleviate this problem, they do not eliminate it. This is the reason why QPNs have hardly been exploited in the past 15 years and very few, if any, practical applications have been reported. The problem is that, until recently, available tools and solution techniques for QPN models were all based on Markov chain analysis, which suffers the well known *state space explosion problem* and limits the size of the models that can be analyzed. This section[3] shows how this problem can be approached by exploiting discrete event simulation for model analysis. We present SimQPN - a Java-based simulation tool for QPNs that can be used to analyze QPN models of realistic size and complexity. While doing this, we propose a methodology for

[3]Originally published in Performance Evaluation Journal, Vol. 63, No. 4-5, S. Kounev and A. Buchmann, *SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation*, pp. 364-394. Copyright Elsevier (2006).

simulating QPN models and analyzing the output data from simulation runs. SimQPN can be seen as an implementation of this methodology.

SimQPN is a discrete-event simulation engine specialized for QPNs. It is extremely lightweight and has been implemented 100% in Java to provide maximum portability and platform-independence. SimQPN simulates QPNs using a sequential algorithm based on the event-scheduling approach for simulation modeling. Being specialized for QPNs, it simulates QPN models directly and has been designed to exploit the knowledge of the structure and behavior of QPNs to improve the efficiency of the simulation. Therefore, SimQPN provides much better performance than a general purpose simulator would provide, both in terms of the speed of simulation and the quality of output data provided.

3.1 SimQPN Design and Architecture

SimQPN has an object-oriented architecture. Every element (for e.g. place, transition or token) of the simulated QPN is internally represented as object. Figure 4 outlines the main simulation routine which drives each simulation run. As already mentioned, SimQPN's internal simulation procedure is based on the event-scheduling approach [Law and Kelton, 2000]. To explain what is understood by event here, we need to look at the way the simulated QPN transitions from one state to another with respect to time. Since only immediate transitions are supported, the only place in the QPN where time is involved is inside the queues of queueing places. Tokens arriving at the queues wait until there is a free server available and are then served. A token's service time distribution determines how long its service continues. After a token has been served it is moved to the depository of the queueing place, which may enable some transitions and trigger their firing. This leads to a change in the marking of the QPN. Once all enabled transitions have fired, the next change of the marking will occur after another service completion at some queue. In this sense, it is the completion of service that initiates each change of the marking. Therefore, we define *event* to be a completion of a token's service at a queue.

SimQPN uses an optimized algorithm for keeping track of the enabling status of transitions. Generally, Petri net simulators need to check for enabled transitions after each change in the marking caused by a transition firing. The exact way they do this, is one of the major factors determining the efficiency of the simulation [Gaeta, 1996]. In [Mortensen, 2001], it is shown how the *locality principle* of colored Petri nets can be exploited to minimize the overhead of checking for enabled transitions. The locality principle states that an occurring transition will only affect the marking on immediate neighbor places, and hence the enabling status of a limited set of neighbor transitions. SimQPN exploits an adaptation of this principle to QPNs, taking into account that tokens deposited into queueing places do not become available for output transitions immediately upon arrival and hence cannot affect the enabling status of the latter. Since checking the enabling status of a transition is a computationally expensive operation, our goal is to make sure that this is done as seldom as possible, i.e., only when there is a real possibility that the status has changed. This translates into the following two cases when the enabling status of a transition needs to be checked:

1. After a change in the token population of an ordinary input place of the transition, as a result of firing of the same or another transition. Three subcases are distinguished:

    (a) Some tokens were added. In this case, it is checked for *newly enabled modes* by considering all modes that are currently marked as disabled and that require tokens of the respective colors added.

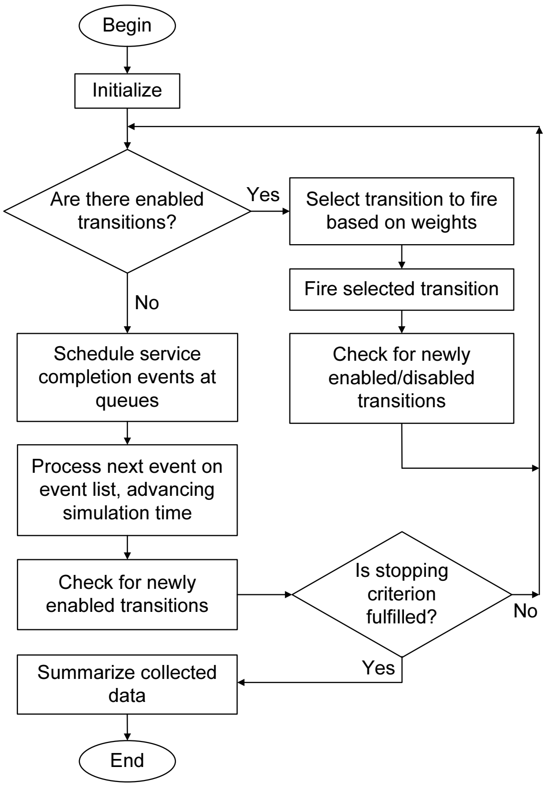
Begin
Initialize
Are there enabled transitions?
Yes
Select transition to fire based on weights
Fire selected transition
Check for newly enabled/disabled transitions
No
Schedule service completion events at queues
Process next event on event list, advancing simulation time
Check for newly enabled transitions
Is stopping criterion fulfilled?
No
Yes
Summarize collected data
End

Figure 4: SimQPN's main simulation routine

(b) Some tokens were removed. In this case, it is checked for *newly disabled modes* by considering all modes that are currently marked as enabled and that require tokens of the respective colors removed.

(c) Some tokens were added and at the same time others were removed. In this case, both of the checks above are performed.

2. After a service completion event at a queueing input place of the transition. The service completion event results in adding a token to the depository of the queueing place. Therefore, in this case, it is only checked for *newly enabled modes* by considering all modes that are currently marked as disabled and that require tokens of the respective color added.

SimQPN maintains a global list of currently enabled transitions and for each transition a list of currently enabled modes. The latter are initialized at the beginning of the simulation by checking the enabling status of all transitions. As the simulation progresses, a transition's enabling status is checked only in the above mentioned cases. This reduces CPU costs and speeds up the simulation substantially.

3.2 Output Data Analysis

SimQPN supports two methods for estimation of the steady state mean residence times of tokens inside the queues, places and depositories of the QPN. These are the well-known *method of independent replications (IR)* (in its variant referred to as *replication/deletion approach*) and the classical *method of non-overlapping batch means (NOBM)*. We refer the reader to [Pawlikowski, 1990; Law and Kelton, 2000; Alexopoulos and Seila, 2001] for an introduction to these methods. Both of them can be used to provide point and interval estimates of the steady state mean token residence time. In cases where one wants to apply a more sophisticated technique for steady state analysis (for example ASAP [Steiger *et al.*, 2005]), SimQPN can be configured to output observed token residence times to files (mode 4), which can then be used as input to external analysis tools. Both the replication/deletion approach and the method of non-overlapping batch means have different variants. Below we discuss some details on the way they were implemented in SimQPN.

Replication/Deletion Approach

We briefly discuss the way the replication/deletion approach is implemented in SimQPN. Suppose that we want to estimate the steady state mean residence time ν of tokens of given color at a given place, queue or depository. As discussed in [Alexopoulos and Seila, 2001], in the replication/deletion approach multiple replications of the simulation are made and the average residence times observed are used to derive steady state estimates. Specifically, suppose that n replications of the simulation are made, each of them generating m residence time observations $Y_{i1}, Y_{i2}, \ldots, Y_{im}$. We delete l observations from the beginning of each set to eliminate the initialization bias. The number of observations deleted is determined through the method of Welch [Heidelberger and Welch, 1983]. Let X_i be given by

$$X_i = \frac{\sum_{j=l+1}^{m} Y_{ij}}{m-l} \qquad i = 1, 2, \ldots, n \tag{1}$$

and

$$\overline{X}(n) = \frac{\sum_{i=1}^{n} X_i}{n} \qquad\qquad S^2(n) = \frac{\sum_{i=1}^{n}[X_i - \overline{X}(n)]^2}{n-1} \tag{2}$$

Then the X_i's are independent and identically distributed (IID) random variables with $E(X_i) \approx \nu$ and $\overline{X}(n)$ is an approximately unbiased point estimator for ν. According to the central limit theorem [Trivedi, 2002], if m is large, the X_i's are going to be approximately normally distributed and therefore the random variable

$$t_n = \frac{[\overline{X}(n) - \nu]}{\sqrt{\frac{S^2(n)}{n}}}$$

will have t distribution with $(n-1)$ degrees of freedom (df) [Hogg and Craig, 1995] and an approximate $100(1-\alpha)$ percent confidence interval for ν is then given by

$$\overline{X}(n) \pm t_{n-1,1-\alpha/2}\sqrt{\frac{S^2(n)}{n}} \tag{3}$$

where $t_{n-1,1-\alpha/2}$ is the upper $(1-\alpha/2)$ critical point for the t distribution with $(n-1)$ df [Pawlikowski, 1990; Trivedi, 2002].

Method of Non-Overlapping Batch Means

Unlike the replication/deletion approach, the method of non-overlapping batch means seeks to obtain independent observations from a single simulation run rather than from multiple replications. Thus, it has the advantage that it must go through the warm-up period only once and is therefore less sensitive to bias from the initial transient. Suppose that we make a simulation run of length m and then divide the resulting observations $Y_1, Y_2, \ldots, Y_m$ into n batches of length q. Assume that $m = n * q$ and let X_i be the sample (or batch) mean of the q observations in the ith batch, i.e.

$$X_i(q) = \frac{\sum_{j=(i-1)q+1}^{(i-1)q+q} Y_j}{q} \qquad i = 1, 2, \ldots, n \tag{4}$$

The mean ν is estimated by $\overline{X}(n) = (\sum_{i=1}^{n} X_i(q))/n$ and it can be shown (see for example [Law and Kelton, 2000]) that an approximate $100(1-\alpha)$ percent confidence interval for ν is given by substituting $X_i(q)$ for X_i in Equations (2) and (3) above.

SimQPN offers two different *stopping criteria* for determining how long the simulation should continue. In the first one, the simulation continues until the QPN has been simulated for a user-specified amount of model time (*fixed-sample-size procedure*). In the second one, the length of the simulation is increased sequentially from one checkpoint to the next, until enough data has been collected to provide estimates of residence times with user-specified precision (*sequential procedure*). The precision is defined as an upper bound for the confidence interval half length. It can be specified either as an absolute value (absolute precision) or as a percentage relative to the mean residence time (relative precision). The sequential approach for controlling the length of the simulation is usually regarded as the only efficient way for ensuring representativeness of the samples of collected observations [Law and Kelton, 1982; Heidelberger and Welch, 1983; Pawlikowski *et al.*, 1998]. Therefore, hereafter we assume that the sequential procedure is used.

The main problem with the method of non-overlapping batch means is to select the batch size q, such that successive batch means are approximately uncorrelated. Different approaches have been proposed in the literature to address this problem (see for example [Chien, 1994; Alexopoulos and Goldsman, 2004; Pawlikowski, 1990]). In SimQPN, we start with a user-configurable initial batch size (by default 200) and then increase it sequentially until the correlation between successive batch means becomes negligible. Thus, the simulation goes through two stages: the first sequentially testing for an acceptable batch size and the second sequentially testing for adequate precision of the residence time estimates (see Figure 5). The parameters n and p, specifying how often checkpoints are made, can be configured by the user.

We use the *jackknife estimators* [Miller, 1974; Pawlikowski, 1990] of the autocorrelation coefficients to measure the correlation between batch means. A jackknife estimator $\hat{J}(k, q)$ of the autocorrelation coefficient of lag k for the sequence of batch means $X_1(q), X_2(q), \ldots, X_n(q)$ of size q is calculated as follows:

$$\hat{J}(k, q) = 2\hat{r}(k, q) - \frac{\hat{r}'(k, q) + \hat{r}''(k, q)}{2} \tag{5}$$

where $\hat{r}(k, q)$ is the ordinary estimator of the autocorrelation coefficient of lag k, calculated from the formula [Pawlikowski, 1990]:

$$\hat{r}(k, q) = \frac{\frac{1}{n-k} \sum_{i=k+1}^{n} [X_i(q) - \overline{X}(n)][X_{i-k}(q) - \overline{X}(n)]}{\frac{1}{n} \sum_{i=1}^{n} [X_i(q) - \overline{X}(n)]^2} \tag{6}$$

and $\hat{r}'(k,q)$ and $\hat{r}''(k,q)$ are calculated like $\hat{r}(k,q)$, except that $\hat{r}(k,q)$ is the estimator over all n batch means, whereas $\hat{r}'(k,q)$ and $\hat{r}''(k,q)$ are estimators over the first and the second half of the analyzed sequence of n batch means, respectively.

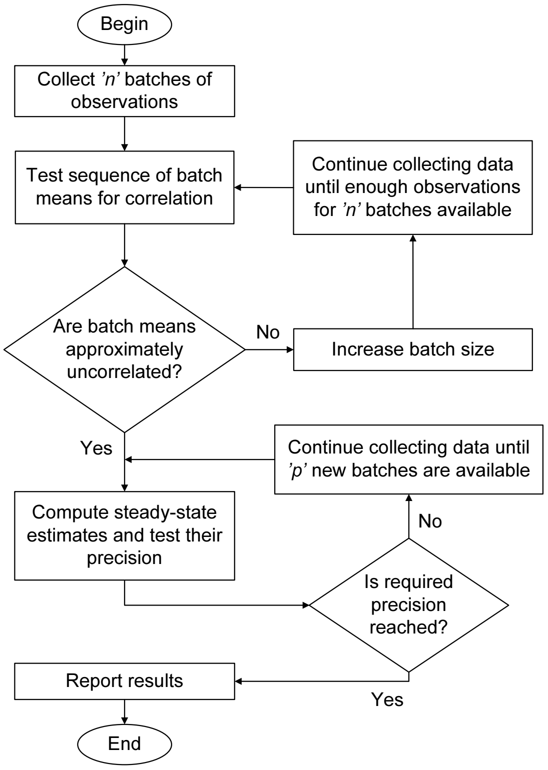
Begin
Collect *'n'* batches of observations
Test sequence of batch means for correlation
Continue collecting data until enough observations for *'n'* batches available
Are batch means approximately uncorrelated?
No
Increase batch size
Yes
Continue collecting data until *'p'* new batches are available
Compute steady-state estimates and test their precision
No
Is required precision reached?
Report results
Yes
End

Figure 5: SimQPN's batch means procedure

We use the algorithm proposed in [Pawlikowski, 1990] to determine when to consider the sequence of batch means for approximately uncorrelated: a given batch size is accepted to yield approximately uncorrelated batch means if all autocorrelation coefficients of lag k ($k = 1, 2, \ldots, L$; where $L = 0.1 * n$) are statistically negligible at a given significance level $\beta_k,\ 0 < \beta_k < 1$. To get an acceptable overall significance level β we assume that

$$\beta < \sum_{k=1}^{L} \beta_k \tag{7}$$

As recommended in [Pawlikowski, 1990], in order to get reasonable estimators of the autocorrelation coefficients, we apply the above batch means correlation test only after at least 100 batch means have been recorded (i.e., $n >= 100$). In fact, by default n is set to 200 in SimQPN. Also to ensure approximate normality of the batch means, the initial batch size (i.e., the minimal batch size) is configured to 200.

SimQPN Validation

We have validated the algorithms implemented in SimQPN by subjecting them to a rigorous experimental analysis and evaluating the quality of point and interval estimates [Kounev and Buchmann, 2006]. In particular, the variability of point estimates provided by SimQPN and the coverage of confidence intervals reported were quantified. A number of different models of realistic size and complexity were considered. Our analysis showed that data reported by SimQPN is very accurate and stable. Even for residence time, the metric with highest variation, the standard deviation of point estimates did not exceed 2.5% of the mean value. In all cases, the estimated coverage of confidence intervals was less than 2% below the nominal value (higher than 88% for 90% confidence intervals and higher than 93% for 95% confidence intervals).

4 Performance Modeling and Analysis of Distributed Systems

Queueing Petri nets are a powerful formalism that can be exploited for modeling distributed systems and analyzing their performance and scalability. However, building models that accurately capture the different aspects of system behavior is a very challenging task when applied to realistic systems. In this section[4], we present a case study in which QPNs are used to model a real-life system and analyze its performance and scalability. In parallel to this, we present a practical performance modeling methodology for distributed systems which helps to construct models that accurately reflect the performance and scalability characteristics of the latter. Our methodology builds on the methodologies proposed by Menascé, Almeida and Dowdy in [Menascé *et al.*, 1994; 1999; Menascé and Almeida, 1998; 2000; Menascé *et al.*, 2004], however, a major difference is that our methodology is based on QPN models as opposed to conventional queueing network models and it is specialized for distributed component-based systems. The system studied is a deployment of the industry-standard SPECjAppServer2004 benchmark. A detailed model of the system and its workload is built in a step-by-step fashion. The model is validated and used to predict the system performance for several deployment configurations and workload scenarios of interest. In each case, the model is analyzed by means of simulation using SimQPN. In order to validate the approach, the model predictions are compared against measurements on the real system. In addition to CPU and I/O contention, it is demonstrated how some more complex aspects of system behavior, such as thread contention and asynchronous processing, can be modeled.

4.1 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 [SPEC, 2004]. Dealers use a Web-based user interface to browse an automobile catalogue, purchase automobiles, sell automobiles and track their inventory. As depicted in Figure 6, SPECjAppServer2004's business model comprises five domains: customer domain dealing with customer orders and

[4]Portions reprinted, with permission, from IEEE Transactions on Software Engineering, Vol. 32, No. 7, *Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets*, pp. 486-502. (c) [2006] IEEE.

interactions, dealer domain offering Web-based interface to the services in the customer domain, manufacturing domain performing "just in time" manufacturing operations, supplier domain handling interactions with external suppliers, and corporate domain managing all dealer, supplier and automobile information.

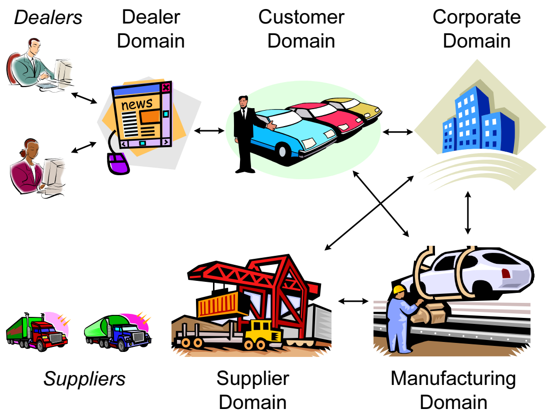

Figure 6: SPECjAppServer2004 business model.

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 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.

4.2 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 Figure 7. 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. In the following, we present a practical performance modeling methodology in a step-by-step fashion showing how each step is applied to the considered scenario.

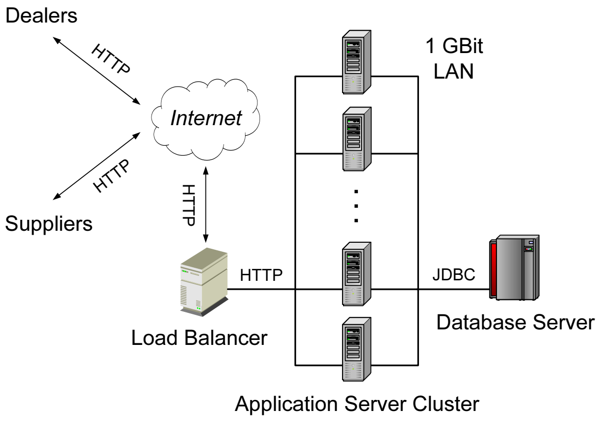

Figure 7: Deployment environment.

4.3 Step 1: Establish Performance Modeling Objectives

Let us assume that under 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% over the next 5 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 peak operating conditions with 6 WebLogic servers. 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 6 WebLogic servers would be enough to ensure that the average response times of business transactions do not exceed half a second. 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.

4.4 Step 2: Characterize the System in its Current State

As shown in Figure 7, 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.

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

4.5 Step 3: Characterize the Workload

Identify the Basic Components of the Workload

As discussed in Section 4.1, the SPECjAppServer2004 benchmark application is made up of three major subapplications - the dealer application, the order entry 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.

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.

Identify the System Components and Resources Used by Each Workload Class

The following hardware resources are used by dealer transactions: CPU of the load balancer machine (LB-C), CPU of an application server in the cluster (AS-C), CPUs of the database

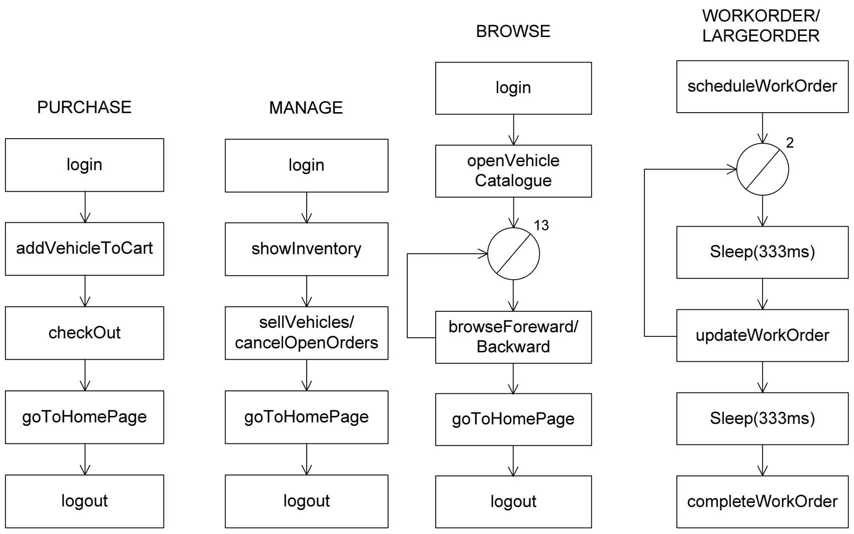

Figure 8: Execution graphs for Purchase, Manage, Browse, WorkOrder and LargeOrder.

server (DB-C), disk drive of the database server (DB-D), Local Area Network (LAN). WorkOrders and LargeOrders use the same resources with 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.

Describe the Inter-Component Interactions and Processing Steps for Each Workload Class

All of the five workload classes identified represent composite transactions. Figure 8 uses 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 inter-component interactions and flow of control during the processing of subtransactions are depicted in Figure 9 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 $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7$. Since most dealer subtransactions do access the database, for simplicity, it is assumed that all of them follow the second path.

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

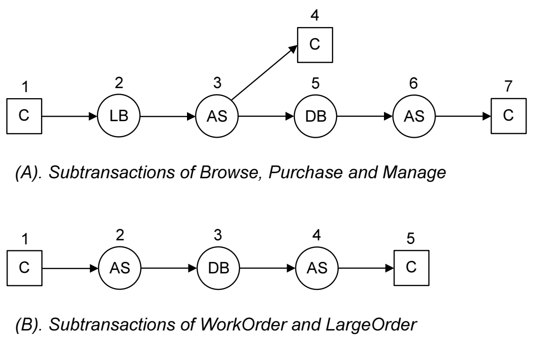
4
C
1
2
3
5
6
7
C
LB
AS
DB
AS
C
(A). Subtransactions of Browse, Purchase and Manage
1
2
3
4
5
C
AS
DB
AS
C
(B). Subtransactions of WorkOrder and LargeOrder

Figure 9: Client/server interaction diagrams for subtransactions.

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 [Menascé and Almeida, 1998]. 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.

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

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.

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 4.3):

- Number of concurrent dealer clients of each type and the average dealer think time.
- 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*).

The concerete values of the above parameters under peak operating conditions were given in Section 4.3. The workload, however, had been forecast to grow by 300% 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.

4.6 Step 4: 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 Figure 8. 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. While characterizing the workload service demands in Section 4.5, 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. 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., relatively small portion of the system workload. Mapping the system components, resources and inter-component interactions to QPN models constructs, we arrive at the model depicted in Figure 10. 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 3 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 man-

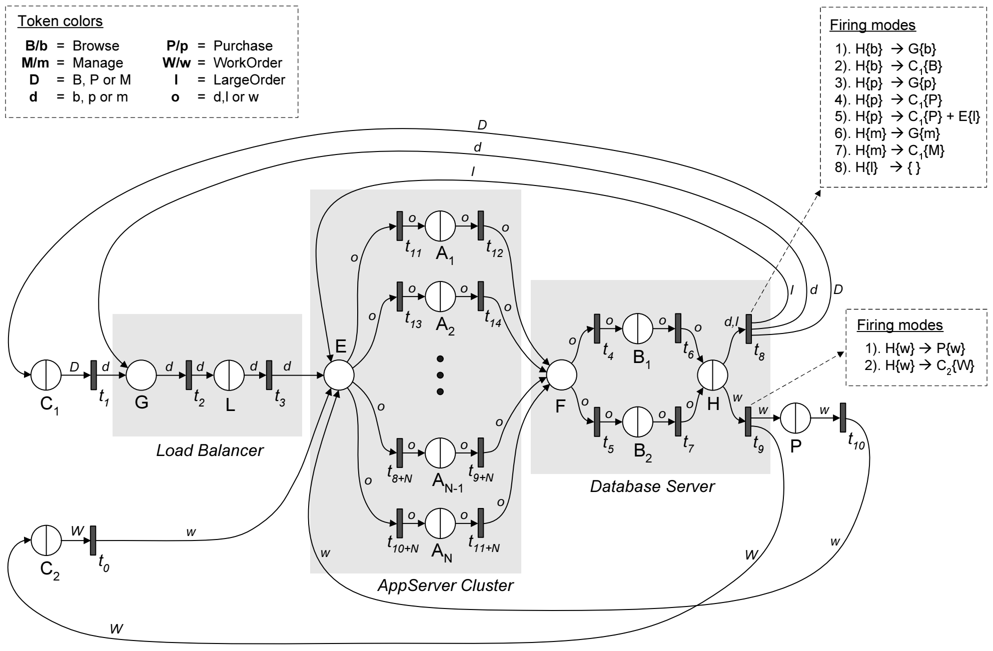

Figure 10: QPN model of the system.

ufacturing 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 subtransaction. The flow of control during 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 $t_{11},\ t_{13},\dots,t_{10+N}$ 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

Table 3: 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.

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 exactly in 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_0, t_1, t_8 and t_9, are defined in such a way that whenever they fire they simply move a token from their input place to their output place. Transitions t_0 and t_1 have similar behavior except that when they remove an upper case token from their input place they deposit the respective lower case token into the output place. 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 multiple of them are 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 Tables 4 and 5, 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 4.5 (Figure 8), 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.

Table 4: 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.

Transition t_8, on the other hand, has eight firing modes as shown in Table 4. According to Section 4.5 (Figure 8), Browse transactions have 17 subtransactions, whereas Purchase and

Table 5: 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.

Manage have only 5. 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. Out of every 5 times a 'p' token arrives in place H and enables transition t_8, the latter has to be fired 4 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% 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 fulfil. 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 4.5 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.

4.7 Step 5: Validate, Refine and/or Calibrate the Model

The model developed in the previous sections was validated by comparing its predictions against measurements on the real system. Two application server nodes were available for the validation experiments. The model predictions were verified for a number of different scenarios under different transaction mixes and workload intensities. The model was analyzed by means of simulation using SimQPN. The method of non-overlapping 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% of the respective mean value. The metrics considered were 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 was 9.3%, for utilization 9.1% and for response time 12.9%. Varying the transaction mix and workload intensity led to predictions of similar accuracy. However, even though the model was deemed valid at this point of the study, as we will see later, the model can lose its validity when it is modified in order to reflect changes in the system.

4.8 Step 6: Use Model to Predict System Performance

In Section 4.3 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 non-overlapping 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% of the respective mean value. Table 7 shows the model predictions for two scenarios under peak conditions with 6 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 6. 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.

Table 6: Load balancer service demands

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

Table 7: Analysis results for scenarios under peak conditions with 6 app. server nodes

	Original Load Balancer			Upgraded Load Balancer		
METRIC	Model (99% c.i.)	Msrd.	Error	Model (99% c.i.)	Msrd.	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%

We now consider the behavior of the system as the workload intensity increases beyond peak conditions and further application server nodes are added. Table 8 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 8. 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.

Table 8: Analysis results for scenarios under heavy load with 8 app. server nodes

	Heavy Load Scenario 1			Heavy Load Scenario 2		
METRIC	Model (99% c.i.)	Msrd.	Error	Model (99% c.i.)	Msrd.	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%

4.9 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 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.

Extending the System Model

In Figure 11, 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.

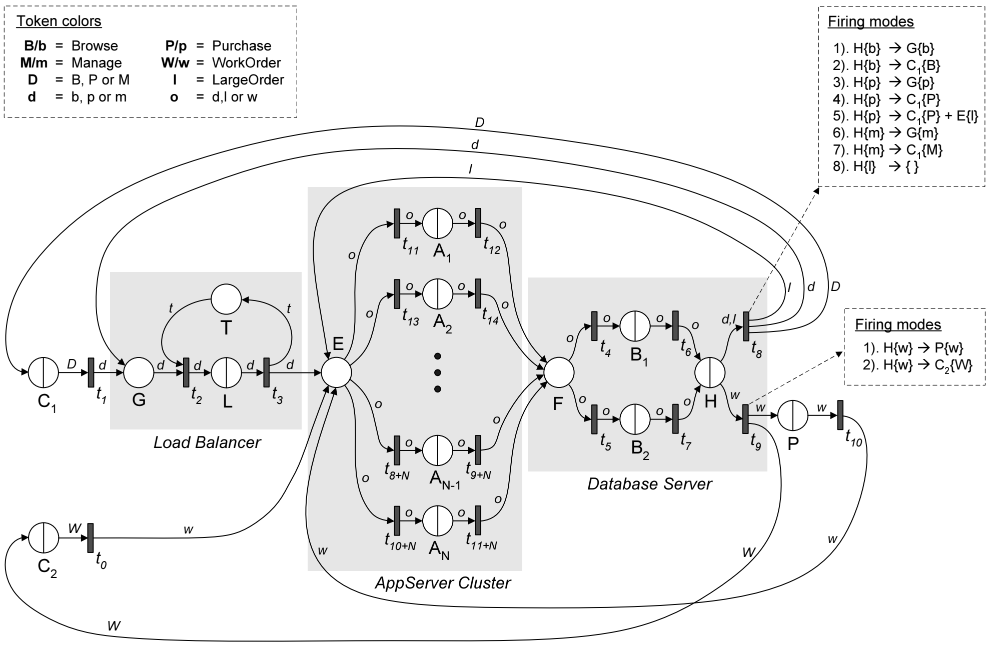

Figure 11: Extended QPN model of the system (capturing thread contention at the load balancer).

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 a FCFS queue. However, simply using a 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 [Bause, 1993] 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 [Bause, 1993], 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 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.

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 9.

Table 9: 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

Table 10: Analysis results for heavy load scenario 3 with 15 and 30 load balancer threads and 8 app. server nodes

	Heavy Load Sc. 3 with 15 Thr.			Heavy Load Sc. 3 with 30 Thr.		
METRIC	Model (99% c.i.)	Msrd.	Error	Model (99% c.i.)	Msrd.	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%

The first scenario has a total of 360 concurrent dealer clients, the second 420. Table 10 compares the model predictions for the first scenario in two configurations with 8 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%. The results for the second scenario look very similar. The CPU utilization of the WebLogic servers and the database server increase to 63% and 52%, respectively, leading to slightly higher response times and lower throughput. The modeling error does not exceed 15.2%. 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.

4.10 Step 7: Analyze Results and Address Modeling Objectives

We can now use the results from the performance analysis to address the goals established in Section 4.3. By means of the developed QPN model, we were able to predict the performance of the system under peak operating conditions with 6 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 to scale the system by adding additional WebLogic servers (see Figure 12). 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.

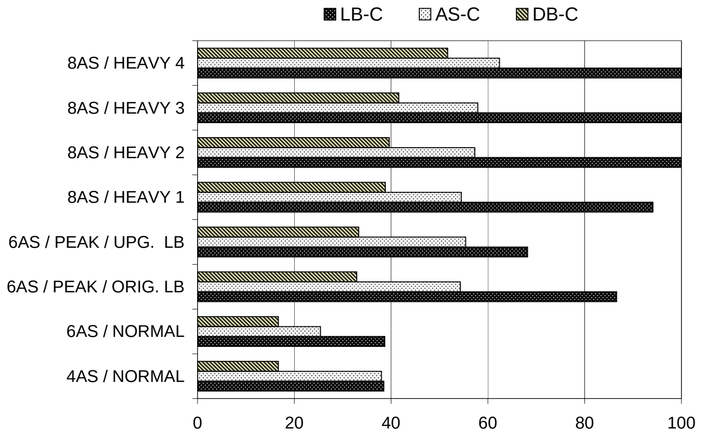

Figure 12: Predicted server CPU utilization in considered scenarios.

5 Concluding Remarks

In this chapter, we showed how QPNs can be exploited for the modeling and performance analysis of distributed systems. We presented a practical performance modeling methodology which helps to construct models of distributed systems that accurately reflect their performance and scalability characteristics. We started by introducing QPNs and discussing the state space explosion problem which is a major hurdle to their practical use. We then showed how the problem can be approached by exploiting discrete event simulation for model analysis. We presented SimQPN - a tool and methodology for simulating QPN models and analyzing the output data from simulation runs.

In the second part of the chapter, we presented a case study of a realistic distributed system, in which we showed how QPN models can be exploited as a powerful performance prediction tool in the software engineering process. The case study was used as an example in order to introduce a practical methodology for performance modeling and analysis of distributed systems. A deployment of the industry-standard SPECjAppServer2004 benchmark was studied, a large and complex application designed to be representative of today's distributed enterprise systems. 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% and was much lower for transaction throughput and resource utilization.

Taking advantage of the modeling power of QPNs, our methodology provides the following important benefits:

1. QPN models allow the integration of hardware and software aspects of system behavior and lend themselves very well to modeling distributed component-based systems.

2. In addition to hardware contention and scheduling strategies, using QPNs one can easily model software contention, simultaneous resource possession, synchronization, blocking and asynchronous processing.

3. By restricting ourselves to QPN models, we can exploit the knowledge of their structure and behavior for fast and efficient simulation using SimQPN. This enables us to analyze models of realistic size and complexity.

4. QPNs can be used to combine qualitative and quantitative system analysis. A number of efficient techniques from Petri net theory can be exploited to verify some important qualitative properties of QPNs. The latter not only help to gain insight into the behavior of the system, but are also essential preconditions for a successful quantitative analysis [Bause, 1993].

5. Last but not least, QPN models have an intuitive graphical representation that facilitates model development.

To support the modeling and analysis of systems using QPNs, we have developed QPME (Queueing Petrinet Modeling Environment) [Kounev *et al.*, 2006]. QPME provides a user-friendly graphical interface enabling the user to quickly and easily construct QPN models. Model analysis is performed using SimQPN. Being implemented as an Eclipse application, QPME runs on all operating systems officially supported by the Eclipse platform. QPME provides a robust and powerful tool for performance analysis making it possible to exploit the modeling power and expressiveness of QPNs to their full potential.

Bibliography

[Alexopoulos and Goldsman, 2004] C. Alexopoulos and D. Goldsman. To Batch Or Not To Batch. *ACM Transactions on Modeling and Computer Simulation*, 14(1):76–114, January 2004.

[Alexopoulos and Seila, 2001] C. Alexopoulos and A. Seila. Output Data Analysis for Simulations. In *Proceedings of the 2001 Winter Simulation Conference, Arlington, VA, USA, December 9-12*, 2001.

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

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

[Bause, 1993] F. Bause. Queueing Petri Nets - A formalism for the combined qualitative and quantitative analysis of systems. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models, Toulouse, France, October 19-22*, 1993.

[Chien, 1994] C. Chien. Batch Size Selection for the Batch Means Method. In *Proceedings of the 1994 Winter Simulation Conference, Lake Buena Vista, FL, USA, December 11-14*, 1994.

[Gaeta, 1996] Rossano Gaeta. Efficient Discrete-Event Simulation of Colored Petri Nets. *IEEE Transactions on Software Engineering*, 22(9), September 1996.

[Heidelberger and Welch, 1983] P. Heidelberger and P. D. Welch. Simulation Run Length Control in the Presence of an Initial Transient. *Operations Research*, 31:1109–1145, 1983.

[Hogg and Craig, 1995] R. V. Hogg and A. F. Craig. *Introduction to Mathematical Statistics*. Prentice-Hall, Upper Saddle River, New Jersey, 5th edition, 1995.

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

[Kounev and Buchmann, 2003] S. Kounev and A. Buchmann. Performance Modelling of Distributed E-Business Applications using Queuing Petri Nets. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software - ISPASS2003, Austin, Texas, USA, March 20-22*, 2003.

[Kounev and Buchmann, 2006] S. Kounev and A. Buchmann. SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 63(4-5):364–394, May 2006.

[Kounev *et al.*, 2006] S. Kounev, C. Dutz, and A. Buchmann. QPME - Queueing Petri Net Modeling Environment. In *Proceedings of the 3rd International Conference on Quantitative Evaluation of SysTems (QEST-2006), Riverside, CA, September 11-14*, 2006.

[Kounev, 2006] S. Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, July 2006.

[Law and Kelton, 1982] A. Law and W. D. Kelton. Confidence Intervals for Steady-State Simulations, II: A Survey of Sequential Procedures. *Management Science*, 28(5):550–562, 1982.

[Law and Kelton, 2000] Averill Law and David W. Kelton. *Simulation Modeling and Analysis*. Mc Graw Hill Companies, Inc., third edition, 2000.

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

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

[Menascé *et al.*, 1994] Daniel A. Menascé, Virgilio A. F. Almeida, and Larry W. Dowdy. *Capacity Planning and Performance Modeling - From Mainframes to Client-Server Systems*. Prentice Hall, Englewood Cliffs, NG, 1994.

[Menascé *et al.*, 1999] D. Menascé, V. Almeida, R. Fonseca, and M. Mendes. A Methodology for Workload Characterization of E-commerce Sites. In *Proceedings of the 1st ACM conference on Electronic commerce, Denver, Colorado, United States*, pages 119–128, November 1999.

[Menascé *et al.*, 2004] Daniel A. Menascé, Virgilio A. F. Almeida, and Lawrence W. Dowdy. *Performance by Design*. Prentice Hall, 2004.

[Miller, 1974] R. G. Miller. The Jackknife: A Review. *Biometrika*, 61:1–15, 1974.

[Mortensen, 2001] Kjeld H. Mortensen. Efficient Data-Structures and Algorithms for a Coloured Petri Nets Simulator. In *Proceedings of the 3rd Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark, August 29-31*, 2001.

[Pawlikowski *et al.*, 1998] K. Pawlikowski, D. Mcnickle, and G. Ewing. Coverage of Confidence Intervals in Sequential Steady-State Simulation. *Journal of Simulation Practice and Theory*, 6(3):255–267, 1998.

[Pawlikowski, 1990] K. Pawlikowski. Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions. *ACM Computing Surveys*, 22(2):123–170, 1990.

[SPEC, 2004] SPEC. SPECjAppServer2004 Documentation. Specifications, April 2004.

[Steiger *et al.*, 2005] N. Steiger, E. Lada, J. Wilson, J. Joines, C. Alexopoulos, and D. Goldsman. ASAP3: a batch means procedure for steady-state simulation analysis. *ACM Transactions on Modeling and Computer Simulation*, 15(1):39–73, 2005.

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