8

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

Samuel Kounev and 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 & 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

Source: Petri Net,Theory and Applications, Book edited by: Vedran Kordic, ISBN 978-3-902613-12-7, pp. 534, February 2008, I-Tech Education and Publishing, Vienna, Austria

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. $I^-, I^+ : P \times T \to \mathbb{N}_0$ *are called backward and forward incidence functions, respectively,*
4. $M_0 : P \to \mathbb{N}_0$ *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 & 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*

$I^-(p,t), I^+(p,t) \in [C(t) \to C(p)_{MS}], \forall (p,t) \in P \times T$[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 & 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 & 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* $t_i \in T_1$ *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 & 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* $t_i \in T_1$ *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.

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

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

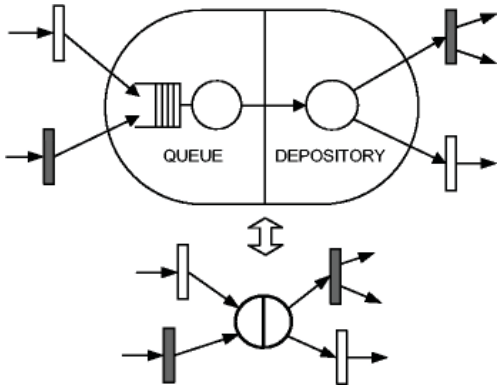

Fig. 1. A queueing place and its shorthand notation.

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. $Q = (Q_1, Q_2, (q_1, ..., q_{|P|}))$ *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*
   - *qi denotes the description of a queue*[2] *taking all colors of C(pi) into consideration, if pi is*
   - a queueing place or equals the keyword 'null', if pi 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* $t_i \in \tilde{W}_1$ or *a firing weight specifying the relative firing frequency due to color c, if* $t_i \in \tilde{W}_2$.

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/l/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*

[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.

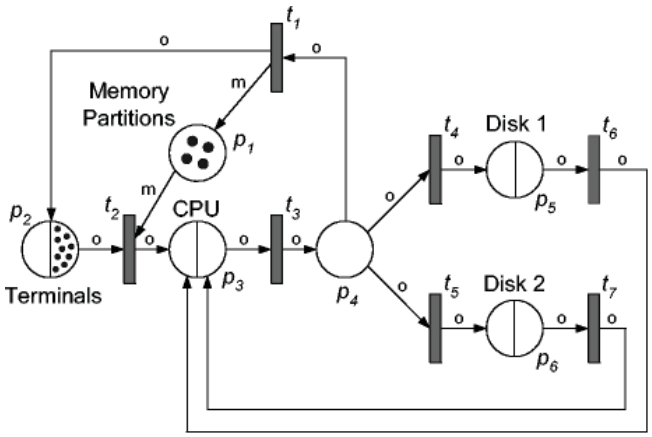

Fig. 2. A QPN model of a central server with memory constraints (reprinted from [Bause & Kritzinger, 2002]).

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 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.

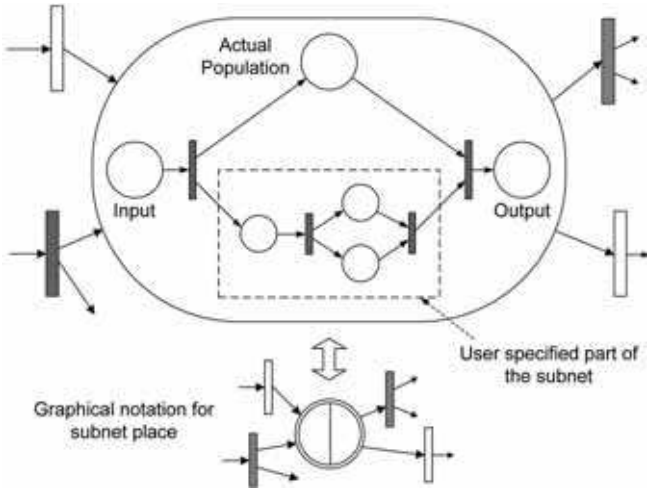

Fig. 3. A subnet place and its shorthand notation.

3. Quantitative analysis of queueing Petri nets

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

[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).

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.

(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.

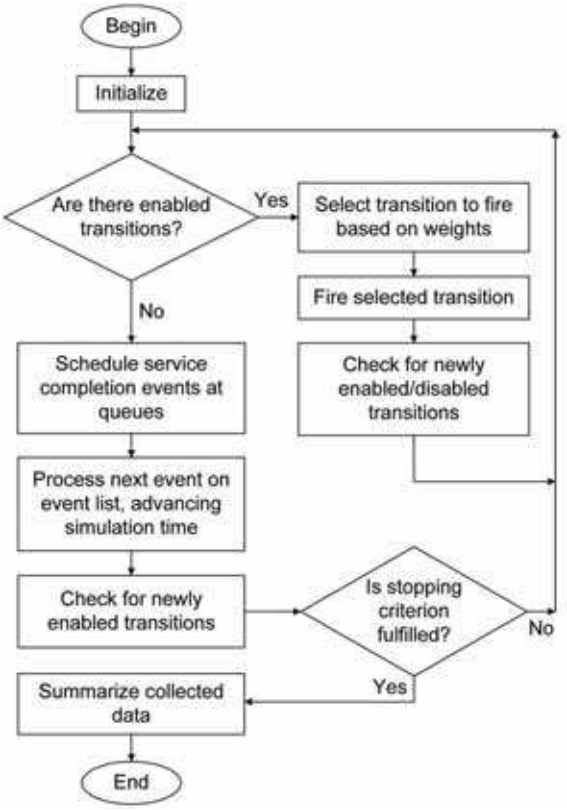

Fig. 4. SimQPN's main simulation routine

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 v 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 $Y_{i1}, Y_{i2}, \bullet\ \bullet\ \bullet, Y_{im}$. 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 Xi be given by

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

and

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

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 v. 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 v 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 $Y_1, Y_2, \bullet\ \bullet\ \bullet, Y_m$ 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 v 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 v 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 & 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.

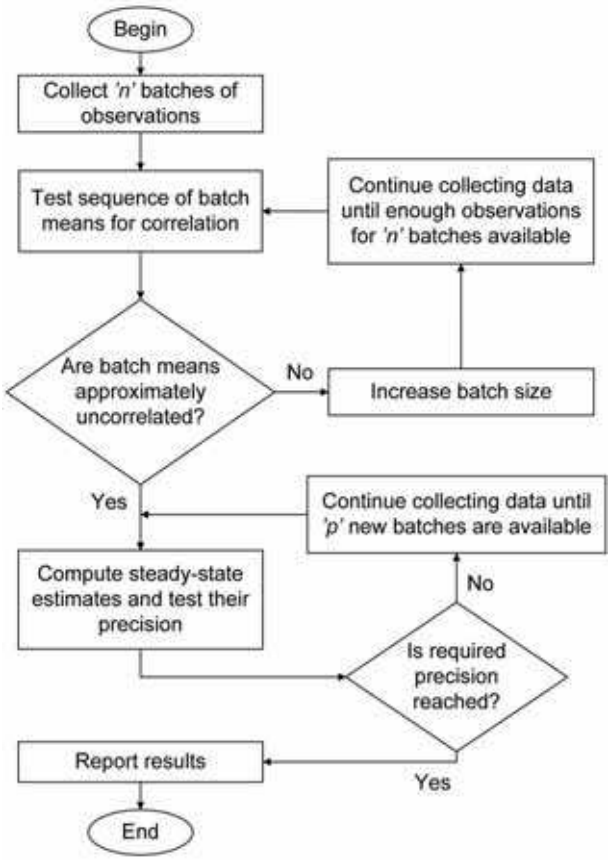

Fig. 5. SimQPN's batch means procedure

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 β_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 & Dowdy in [Menascé *et al,* 1994; 1999; Menascé & 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

[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.

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 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.

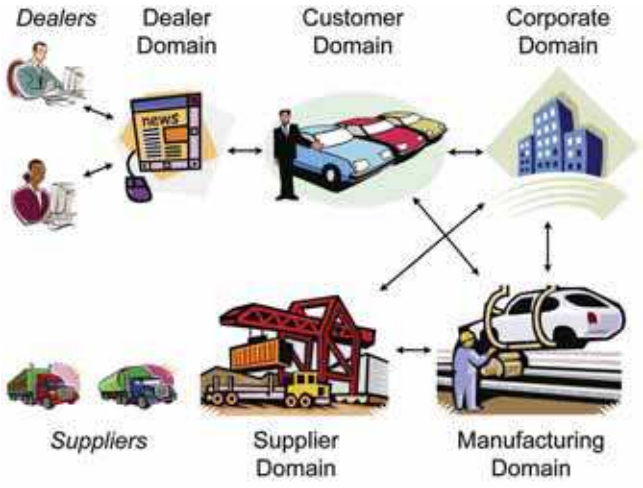

Fig. 6. SPECjAppServer2004 business model.

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.

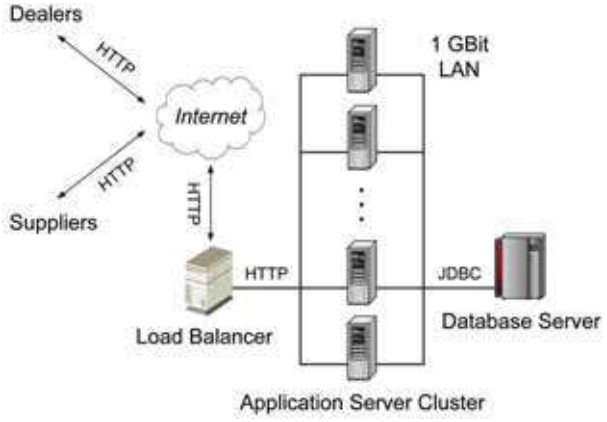

Fig. 7. Deployment environment.

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.

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

Table 1. System component details

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 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.

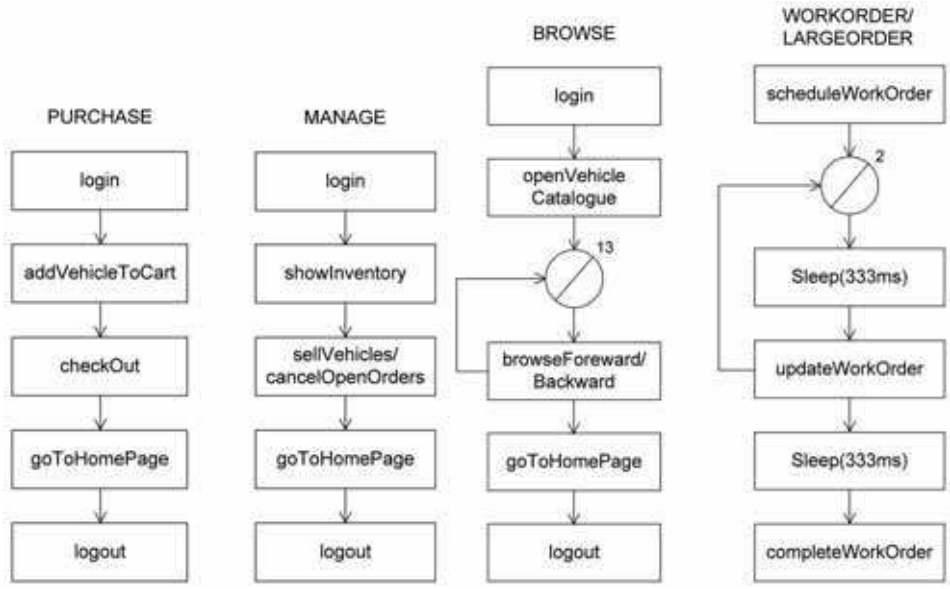

Fig. 8. Execution graphs for Purchase, Manage, Browse, WorkOrder and LargeOrder.

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→2→3→4, whereas dealer subtransactions that access

the database (e.g., showlnven-tory) follow the path 1→2→3→5→6→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 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 [Menasce 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.

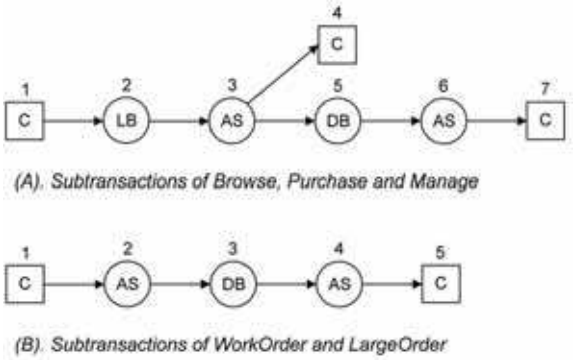

Fig. 9. Client/server interaction diagrams for Subtransactions.

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 2. Workload service demand parameters

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