An Analytical Approach to Performance Analysis of an Asynchronous Web Server

U. Praphamontripong, S. Gokhale
Dept. of CSE
Univ. of Connecticut
Storrs, CT 06269
ssg@engr.uconn.edu

Aniruddha Gokhale
Dept. of EECS
Vanderbilt Univ.
Nashville, TN 37235
a.gokhale@vanderbilt.edu

Jeff Gray
Dept. of CIS
U. of Alabama at Birmingham
Birmingham, AL 35294
gray@cis.uab.edu

Abstract

Concurrency can be implemented in a Web server using synchronous and asynchronous mechanisms provided by the underlying operating system. Compared to the synchronous mechanisms, asynchronous mechanisms are attractive because they provide the benefit of concurrency while alleviating much of the overhead and complexity of multi-threading. The Proactor pattern in middleware, which effectively encapsulates the asynchronous mechanisms provided by an operating system, can be used to implement a high performance Web server.

The performance expectations imposed on a Web server make it necessary to analyze its performance prior to deployment. While performance of a server can be measured after implementation, design-time performance analysis, conducted early in the life cycle, can also enable informed configuration and provisioning choices. A model-based approach can be used for such design-time performance analysis. In this paper we present a queuing model of an asynchronous Web server implemented using the Proactor pattern. We discuss the implementation of the queuing model using the Stochastic Reward Net (SRN) modeling paradigm. A model decomposition strategy along with its SRN implementation to enable the application of the model to practical Web servers is then described. We demonstrate the use of the model to guide key provisioning and configuration decisions using several examples.

1 Introduction and motivation

Within a relatively short duration since its advent, the World Wide Web (WWW) has become an important source of information and services. Initially, users were attracted to the WWW primarily due to the convenience, flexibility, ease of use and low costs associated with the use of these services. However, as the

prevalence of WWW in business and critical domains grows, it is becoming evident that WWW services must be offered with superior performance in order to retain existing users and attract new ones [31].

A central component of any WWW service is a Web server. Modern Web servers have to process millions of client requests on a daily basis. In order to fulfill such high workload demands, it is inevitable that modern Web servers be equipped with the capability to process multiple requests concurrently. Concurrency may be implemented in a Web server using the synchronous or asynchronous capabilities provided by the underlying operating system. Although multi-thread and multi-process Web server architectures [16] which rely on the synchronous capabilities are commonly used, the asynchronous mechanisms may be attractive because they provide the benefit of concurrency while alleviating much of the overhead and complexity of multi-threading. The Proactor pattern in middleware [25], which effectively encapsulates the asynchronous mechanisms supported by the operating system, can be used to implement a high performance Web server.

Due to the high performance expectations associated with a WWW service, it is imperative that service performance be analyzed prior to deployment. Although performance can be measured once the service is implemented, it is often too late and expensive to take corrective action at this stage if it is discovered that the target performance cannot be met. It is thus cost-effective and advantageous to conduct performance analysis earlier in the life cycle at design time. Model-based analysis is an attractive approach to conduct such design-time performance analysis.

In this paper we describe a model-based approach for the design-time performance analysis of a Web server which implements concurrent processing capabilities using the asynchronous mechanisms encapsulated in the Proactor pattern. We capture the characteristics of the Proactor pattern that are relevant from a performance perspective into a queuing model. We then discuss how the queuing model can be implemented using the Stochastic Reward Net (SRN) modeling paradigm. We then describe a model decomposition strategy to enable the use of the model to estimate the performance metrics in practical scenarios. The SRN implementation of the model decomposition strategy is also discussed. We illustrate how the model can be used to guide configuration and provisioning decisions with several examples.

The balance of the paper is organized as follows: Section 2 provides an overview of the Proactor pattern. A brief background on SRNs is presented in Section 3. Section 4 describes the performance analysis methodology. Section 5 illustrates the potential of the methodology with examples. Section 6 summarizes the related research. Section 7 offers concluding remarks and directions for future research.

2 Proactor pattern

In this section we provide an overview of the Proactor pattern. We also discuss the advantages of implementing a Web server using the Proactor pattern.

2.1 Proactor description

The Proactor pattern is a software architectural pattern for event handling, which is used to describe how to initiate, receive, demultiplex, dispatch and process events in network systems [25]. It has been primarily developed to support many simultaneous user requests. Its main purpose is to improve the performance of an event-driven application that receives and processes multiple events asynchronously. Conceptually, this pattern simplifies asynchronous operations by integrating the demultiplexing of completion events and the dispatching of the corresponding event handlers. The general idea of the Proactor pattern is to wait for an event to occur and then initiate the appropriate operation. Once the event starts execution, other events may be initiated and processed. When the event finishes execution, the Proactor demultiplexes the completion event and dispatches it to an appropriate event handler for subsequent processing of the results of the operation.

To implement the Proactor pattern (considering the arrivals of events, each of which requires a single operation to complete), when an event arrives, the application's entity called an initiator starts an appropriate asynchronous operation. The pattern then registers the event with an associated event handler and event dispatcher with the Asynchronous Operation Processor (AOP). Then an initiator invokes the registered asynchronous operation on the AOP. An asynchronous operation is executed without blocking its caller's thread of control. As a result, the caller can perform other operations. That is, the operation and the initiator can run independently and the initiator can invoke a new asynchronous operation while others continue executing concurrently. If an operation must wait for the occurrence of an event, such as a connection request generated by a remote application, its execution will be deferred until the event arrives. In this paper, however, we consider only those cases where operations are processed independently and do not wait for the occurrence of other events. Once the operation is complete, the AOP retrieves information corresponding to an event handler and a dispatcher, and generates a completion event containing the results of the asynchronous operation. The Proactor then inserts the completion event along with the retrieved information into the completion event queue. It then removes the completion event from the completion event queue and demultiplexes and dispatches the event to the event handler associated with the asynchronous operation. Subsequently, the event handler processes the results of the asynchronous operation and calls back to the application.

2.2 Proactor advantages

There are several advantages to implementing a Web server using the Proactor pattern [25]. These include:

- The Proactor pattern executes each asynchronous operation independently; thus, each service that a Web server provides can be processed separately. Accordingly, a particular demultiplexer and a dispatcher used for each completion event associated with an asynchronous operation can be implemented, managed, and treated independently. As a consequence, the implementation of the Web server is decomposed and decoupled, and hence is more manageable.
- Structuring the demultiplexing and dispatching of completion events simplifies the development process of a Web server, which normally requires asynchronous operations.
- Once an asynchronous operation is initiated, the thread that initiated the operation becomes available to service additional requests.
- Since the asynchronous operations are processed concurrently and the completion events associated with the operations are demultiplexed and dispatched asynchronously, the operations are executed without waiting for the completion of the previous ones. Multiple client requests can be processed simultaneously, which may improve server performance.

3 Overview of SRNs

This section provides an overview of the Stochastic Reward Net (SRN) modeling paradigm which is used to implement the performance model of the Proactor pattern. The details of SRNs can be obtained from elsewhere [20].

A SRN is a directed graph, which contains two types of nodes: *places* and *transitions*. A directed arc connecting a place (transition) to a transition (place) is called an *input (output) arc*. Arcs are associated with a positive integer called the *multiplicity*. Places can contain *tokens* that move from one place to another through transitions. A transition is enabled when each of the places connected to it by its input arc have at least the number of tokens equal to the multiplicity of those arcs. When an enabled transition fires, a number of tokens equal to the input arc multiplicity is removed from each of the corresponding input places, and a number of tokens equal to the output arc multiplicity is deposited in each of the corresponding output places. A SRN may also include an *inhibitor arc*, which can also have a multiplicity associated with it. An inhibitor arc inhibits the transition it is connected to if the place it is connected to at its other end has a number of tokens equal to at least its multiplicity. The state of a SRN with P places is represented by a

vector $(m_1, m_2, \cdots, m_p)$ called the *marking* of the SRN, where m_i is the number of tokens in place i. A SRN marking with at least one immediate transition enabled is called a *vanishing marking*, and a marking with no immediate transitions enabled is called a *tangible marking*. A reward rate may be associated with each tangible marking of a SRN. The tangible markings of a SRN and the rates of transition among them are equivalent to the corresponding states and state transitions of an underlying continuous time Markov chain (CTMC) [29]. Hence, a SRN can be mapped into an equivalent Markov reward model (MRM) [20], automatically using software tools such as SPNP [7]. SRN models allow the concise specification of various reward functions. To extend the power of specification, a SRN may also include the specification of *enabling* (or *guard*) *functions* for each transition. The transition is enabled only if the enabling function returns 1.

SRNs substantially extend the modeling power of Generalized Stochastic Petri Nets (GSPNs) [24], which are an extension of Petri nets [19]. SRNs represent a powerful modeling technique with concise specification and form closer to a designer's intuition. As a result, it is also easier to transfer the results obtained from solving the models and interpret them in terms of the entities that exist in the system being modeled. SRNs have been extensively used for performance, reliability and performability analysis of a variety of systems including cluster systems, polling systems, and wireless networks [29].

4 Performance analysis methodology

In this section we discuss the performance analysis methodology for a Proactor-based asynchronous Web server. We first discuss the characteristics of the Web server, followed by the desired performance metrics. Subsequently, we describe the performance model of the Web server and the SRN implementation of the model, followed by the model decomposition strategy and its implementation.

4.1 Web server characteristics

A Web server employs the request/reply paradigm, using the HTTP protocol to communicate between itself and the clients (Web browsers). The clients' requests are specified in a HTTP message, which may also include the operation to be performed and its location. We consider a scenario where a Web server provides m types of services. As soon as a request arrives at the Web server, a corresponding operation is assigned, initiated, and executed. The completion of these operations is handled in a common queue regardless of the request type. The completion events are demultiplexed by the Proactor and dispatched to an appropriate completion event handler. The completion events are further processed by the completion event handlers. As an instance, consider a Web server which supports a read request. When a client issues a read request

for a particular file, an HTTP handler can be used to initiate a read operation. After the read operation is complete, an HTTP handler, which now acts as a completion event handler, further processes the request by issuing a write operation to transfer the file to the client.

From the point of view of performance analysis, the Web server has the following characteristics:

- The server receives m types of client requests.
- To service these requests, appropriate asynchronous operations are assigned and executed with a pool of handlers for each request type registered with the Proactor. If an incoming request finds that all the event handlers for that request type are busy, the request is rejected. To distinguish between handlers that handle asynchronous operations and handlers that deal with completion operations, the former are simply referred to as event handlers, while the latter are called completion event handlers.
- The completion operations of all the types of requests, referred to as completion events, are queued in a single completion event queue.
- The completion events are dequeued by the Proactor on a first-in, first-out basis.
- Each request type has a separate queue holding completion event operations which are then processed by the completion event handler registered with the Proactor.
- Each request type has a single completion event handler to process the completion events.

4.2 Performance metrics

In this section we present the performance metrics for each request type. We also discuss the practical significance of these metrics.

1. *Expected throughput (T_i):* It is the average processing rate of the Web server.
2. *Expected loss probability (L_i):* It is the average probability that an incoming request will be rejected because an event handler is unavailable.
3. *Expected response time (R_i):* It is the average time taken by the Web server to serve a client request.
4. *Expected busy handlers (B_i):* It is the average number of busy event handlers. This can be used to guide provisioning decisions regarding the sizes of the event handler pools for a given load.
5. *Expected queue lengths (Q and Q_i):* It is the average number of completion events in the common and separate completion event queues.

Typically, a service provider has to satisfy dual and conflicting objectives; namely, offer superior service performance while keeping the cost minimal. From a client's perspective, service performance is superior, if the requests are handled at the same rate as they are presented to the server and with negligible loss and acceptable response time. From a provider's perspective, provisioning of adequate resources (resources consist of event handlers in the the event handler pools and buffer space for queues) is a way to provide superior performance in a cost-effective manner. However, achieving the correct level of provisioning must be balanced: over-provisioning would guarantee good performance, but will be expensive; under-provisioning would have low cost, but would sacrifice performance. Thus, metrics #1 through #3 are important for a user, whereas metrics #4 and #5 are important for a provider.

4.3 Performance model

In this section we first describe the performance model of a Proactor-based asynchronous Web server, followed by the implementation of the model using the SRN modeling paradigm.

4.3.1 Description of the model

We assume that the requests of each type arrive according to a Poisson distribution, with λ_i denoting the arrival rate of type i requests. The size of the event handler pool of type i requests is denoted N_i. The service time of an asynchronous operation for each request type follows an exponential distribution, with the parameter of type i request denoted μ_i. The capacity of the common completion queue is denoted C and the capacity of the separate completion queue of type i requests is denoted C_i. The demultiplexing time of the Proactor is exponentially distributed with parameter κ. The service times of the completion event handlers are exponentially distributed, with the service time of event handler i denoted γ_i.

Figure 1 shows the queuing model of a Proactor-based Web server. Each event handler pool which handles asynchronous operations for a single request type is modeled as a multi-server processing station with no queuing. These servers feed completion events to the common completion event queue and they block if there is no space available in the common completion queue. The operation of demultiplexing the completion events is conducted by a server, which accepts the completion events from the common completion queue and dispatches them to the appropriate separate completion queue depending on the request type that generated the completion event. Because the scheduling used for demultiplexing is first-in, first-out, the demultiplexing operation blocks if the completion queue to which the event is to be dispatched is full. For example, if the current completion event to be demultiplexed and dispatched is of type i and the type i completion queue has C_i completion events, the demultiplexing operation blocks. The completion event queue

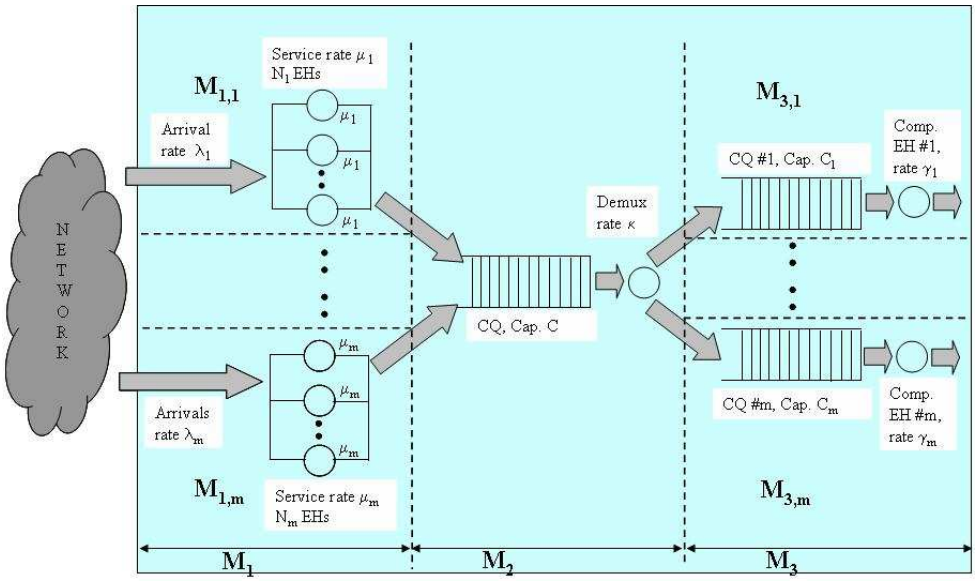

Figure 1: Queuing model of an asynchronous Web server

of each event type feeds the corresponding completion handler which completes the processing. Thus, the three steps involved in fulfilling a single client request are: (i) asynchronous operation, (ii) completion event demultiplexing and (iii) dispatching and completion event handling.

4.3.2 Implementation of the model

Figure 2 shows the implementation of the queuing model in Figure 1 using the SRN modeling paradigm. We discuss how the three steps involved in fulfilling a client request are represented by the SRN model.

I: Asynchronous operation:

The asynchronous operation is performed separately for each request type. For a request type i, place S_i represents the pool of event handlers, transition A_i represents the arrival of requests and transition Sr_i represents the asynchronous operation of the requests by the event handler pool. The firing rate of transition A_i is λ_i. Transition Sr_i has a marking-dependent firing rate and is equal to $s_i \times \mu_i$, where s_i is the number of tokens in place S_i. The maximum number of tokens in place S_i is equal to the pool size N_i, which results in the maximum firing rate of transition Sr_i to be $N_i \times \mu_i$. The presence of N_i tokens in place S_i indicates that all the event handlers in the pool are busy, which should cause an incoming request to be rejected. This is achieved by the inhibitor arc from place S_i to transition A_i with multiplicity N_i.

II: Completion event demultiplexing and dispatching:

The event demultiplexing and disptaching step is handled commonly for all the request types. Place CQ represents the common completion event queue into which a token is deposited by the firing of transitions Sr_is. Inhibitor arcs from place CQ to transitions Sr_is prevent the firing of transition Sr_is when there are C tokens in the completion queue. The firing of transition Int dequeues a completion event from the queue and begins its demultiplexing, which is represented by a token in place DQ. To ensure that exactly one completion event is demultiplexed at a time, the firing of transition Int is prevented by an inhibitor arc from place DQ. Once a demultiplexing operation is complete, transition Sd fires and deposits a token in an intermediate place DP, which serves to dispatch the completion event to one of the separate completion event queues. The dispatching is achieved by probabilistic firing of transitions Sd_is. The transition probability assigned to transition Sd_i is such that its likelihood of firing is the same as the probability with which an event fed into the completion event queue is of type i. This probability is given by the ratio of the firing rate of transition Sr_i to the sum of the firing rates of transition Sr_is.

III: Completion event handling:

Similar to the first step, the third step is conducted separately for each type of event. For an event type i, place CQ_i represents the separate completion event queue and E_i represents the completion event handler. Transition Sc_i represents the handling of a completion event. The firing rate of transition Sc_i is γ_i. The firing of the immediate transition Sp_i initiates the service of the completion event by the completion event handler. Thus, Sp_i can fire only when there is no token in place E_i, which indicates that the completion event handler is available. This is achieved by the inhibitor arc from place E_i to transition Sp_i. To represent the blocking of the demultiplexing operation when the separate completion event queue of an event type which is at the head of the common completion queue is full, inhibitor arcs are added from places CQ_is to place CQ with multiplicities C_is. We note that this represents a pessimistic scenario, since it blocks the demultiplexing operation when any one of the separate completion queues are full, regardless of the event type at the head of the common completion queue.

4.4 Model decomposition

The SRN implementation of the queuing model shown in Figure 2 can be solved as is using the numerical solvers in the Stochastic Petri Net Package (SPNP) [7] to estimate the performance metrics for a given choice of parameters. However, for the pool and queue sizes used in practical Web servers (for example, in the Apache Web server the queue size is set to 3000 and the number of threads is set to 25 by default [15]), state space explosion would make solving the SRN model infeasible. To alleviate the state space explosion

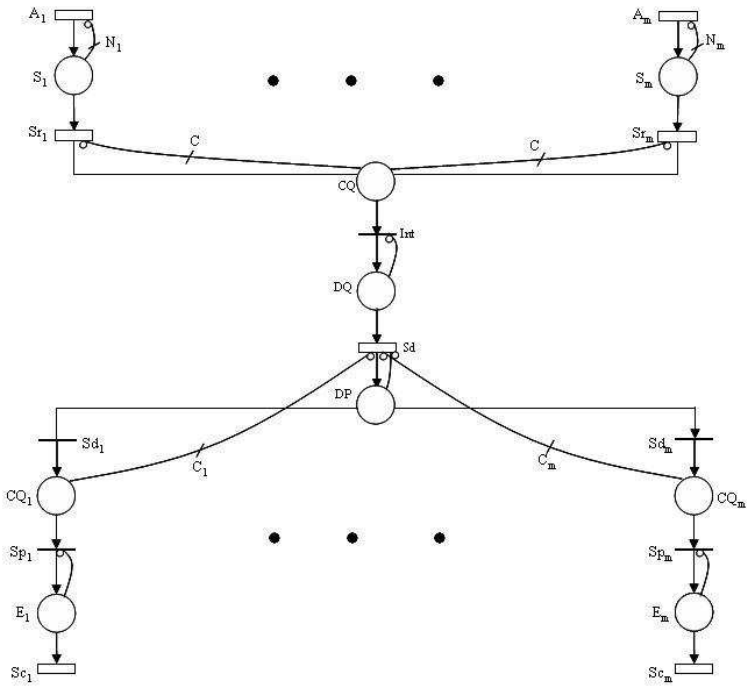

Figure 2: SRN implentation of the queuing model

issue, we describe a model decomposition strategy in this section. We then discuss how the decomposition strategy can be implemented in the SRN model.

4.4.1 Description of the strategy

The model decomposition strategy consists of hierarchically partitioning the overall model into sub-models. The sub-models are then solved and their results are combined to obtain the performance estimates.

The queuing model is partitioned into three sub-models, namely, M_1, M_2 and M_3 as shown in Figure 1. Sub-model M_1 comprises of the event handler pools, sub-model M_2 captures the demultiplexing and dispatching of the completion events and sub-model M_3 represents the handling of the completion events. In sub-model M_1, the different types of requests are handled independently of each other by their respective pools of event handlers. As a result, sub-model M_1 can be partitioned further into m lower level sub-models, denoted $M_{1,1}, \ldots, M_{1,m}$ for each of the m request types. Similarly, in sub-model M_3, the queuing and the handling of the completion events is handled independently for the different event types, due to which it can also be partitioned further into m lower level sub-models, denoted $M_{3,1}, \ldots, M_{3,m}$. For type i requests, the parameters of sub-model $M_{1,i}$ are the size of the event handler pool and the service rate of each event handler. These parameters will impact the loss probabilities and the number of busy event handlers. The solution of sub-model $M_{1,i}$ will provide the average rate at which type i requests are processed by the event

handler pool, denoted $\alpha_{1,i}$. $\alpha_{1,i}$s serve as inputs to the second sub-model M_2. The total input rate to the second sub-model, denoted α_2 is given by:

$$\alpha_2 = \alpha_{1,1} + \alpha_{1,2} + \cdots + \alpha_{1,m} \tag{1}$$

The rate at which the Proactor demultiplexes a single completion event and the capacity of the common completion queue are the parameters of second sub-model M_2. These parameters will impact the average queue length of the completion queue, the total demultiplexing time τ_2 for a completion event (queuing time plus the demultiplexing time) and the effective demultiplexing rate, denoted δ. The input rate to model $M_{3,i}$, denoted $\eta_{3,i}$, is given by:

$$\eta_{3,i} = \frac{\alpha_{1,i}}{\alpha_{1,1} + \cdots + \alpha_{1,m}}\delta \tag{2}$$

Equation (2) scales the effective demultiplexing rate δ by the likelihood of a type i event being fed into the common completion queue to determine the rate at which completion events are demultiplexed into the separate completion queue of type i requests. The parameters of sub-model $M_{3,i}$ include the service rate of the completion event handler and the capacity of the separate completion event queue. These parameters will influence the total time taken to process a completion event denoted $\tau_{3,i}$ (queuing time plus the completion event handling time) and the throughputs of the Web server.

Table 1 summarizes the inputs, parameters and outputs of the different sub-models. In summary, estimates of average loss probabilities and the average number of busy handlers are obtained from the solution of the sub-models $M_{1,1}, \ldots, M_{1,m}$, the total demultiplexing time and the average queue length of the completion queue are provided by the solution of sub-model M_2 and the handling times of the completion events, queue lengths of the separate completion queues and the throughputs are obtained from the solution of sub-models $M_{3,1}, \ldots, M_{3,m}$.

The end-to-end response time of a type i request obtained by adding the time spent by the request in the three processing steps is given by:

$$R_i = \tau_{1,i} + \tau_2 + \tau_{3,i} \tag{3}$$

In Equation (3), $\tau_{1,i}$ is the contribution of sub-model $M_{1,i}$ to the response time and is given by $1/\mu_i$. τ_2 is the contribution of sub-model M_2 (demultiplexing and queuing in the common completion queue) to the response time. Since the demultiplexing is conducted on a first-in, first-out basis and one at a time, the contribution of sub-model M_2 to the response time is the same for all the types of requests. $\tau_{3,i}$ is the contribution of sub-model $M_{3,i}$ (queuing in the separate completion queue and completion event handling) to the response time.

Table 1: Inputs, parameters and outputs of sub-models

Sub-model	Input	Parameter	Output
$M_{1,i}$	Arrival rate (λ_i)	Pool size (N_i)	Loss probability (L_i)
		Service rate (μ_i)	Busy handlers (B_i)
			Avg. processing rate ($\alpha_{1,i}$)
M_2	Input rate (α_2)	Capacity (C)	Queue length (Q)
		Demux. rate (κ)	Total demux. time (τ_2)
			Eff. demux. rate (δ)
$M_{3,i}$	Input rate ($\eta_{3,i}$)	Capacity (C_i)	Queue length (Q_i)
		Comp. event hand. rate (γ_i)	Total hand. time ($\tau_{3,i}$)
			Throughput (T_i)

Typically, using a model decomposition strategy only approximate performance estimates can be obtained. This is because of the errors and inaccuracies that are introduced at the interfaces of the sub-models and the propagation of these errors and inaccuracies to the other sub-models. These inaccuracies arise due to the implicit assumptions that the decoupling and partitioning of the different pieces of the model is based upon. In this case, at the interface of sub-models M_1 and M_2, the decomposition strategy assumes that the event handlers do not block due to the lack of space in the common completion event queue. This will hold if the capacity of the common queue is sufficiently large to absorb any short-term spikes or bursts and the demultiplexing rate is greater than the rate at which the completion events are fed into the common queue to ensure long-term stability. Similarly, at the interface of sub-models M_2 and $M_{3,i}$s, the decomposition scheme assumes that the demultiplexing operation does not block due to the lack of space in the separate completion event queues. Once again, by provisioning separate queues with large capacities and ensuring that the service rate of each completion event handler is greater than the rate at which the completion events are fed into its queue, it can be ensured that this assumption holds. The second factor which gives rise to approximate estimates also arises at the interfaces between the sub-models, when the outputs produced from one sub-model are fed as inputs to other sub-models. Although it is possible to obtain the average values of the output of one sub-model (and hence the average values of the inputs to the next sub-model), it is usually difficult to determine the distributions of these outputs (inputs). This makes it necessary to assume the distribution of the inputs, which may result in approximate estimates. For example, in the queuing model of the Proactor-based server, where the processing rates of sub-models $M_{1,1}, \ldots, M_{1,m}$ serve as the input rates to

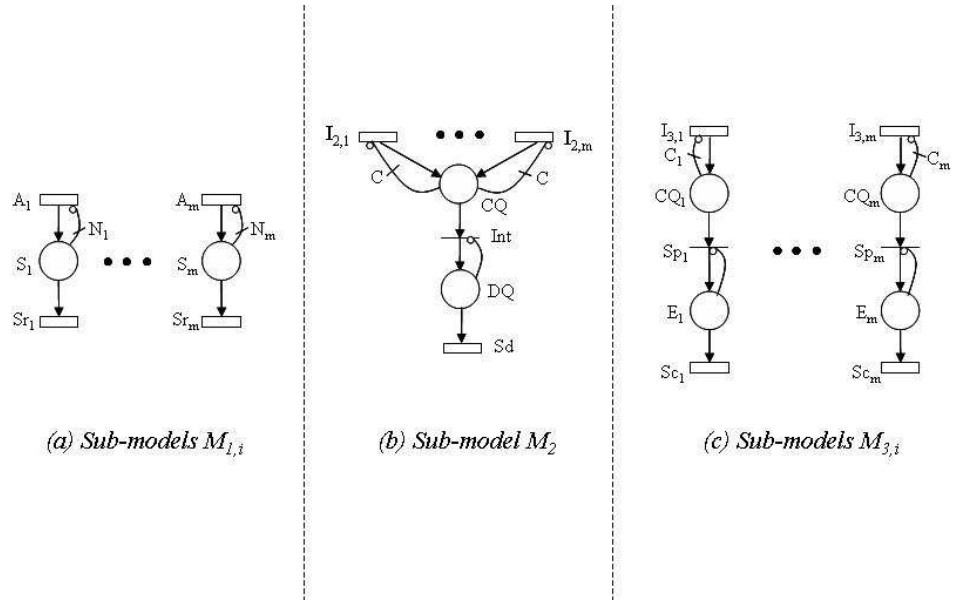

Figure 3: Model decomposition in SRN

sub-model M_2, the average processing rates and hence the average input rates can be determined, but it may not be possible to determine the actual distribution.

Although exact performance estimates can be rarely obtained using a model decomposition scheme, the approximate performance estimates that can be obtained usually provide sufficient information for design-time analysis, where the primary purpose is to determine the range of parameter choices for which the performance is acceptable for an expected load. Such information can be used to guide provisioning and configuration decisions.

4.4.2 Implementation of the strategy

The model decomposition strategy can be implemented by partitioning the SRN model in Figure 2 as shown in Figure 3. The figure indicates that additional transitions and arcs are needed in sub-models M_2 and $M_{3,i}$s to facilitate model decomposition.

In sub-model M_2, transitions $I_{2,i}$ represent inputs to the common completion event queue, which are provided by the outputs of sub-models $M_{1,i}$s. The firing rate of transition $I_{2,i}$ is the same as the firing rate of transition Sr_i and is given by $\alpha_{1,i}$. Inhibitor arcs from place CQ to transitions $I_{2,i}$s with multiplicity C prevent their firing when the common completion queue is full. The effective demultiplexing rate δ is the rate at which transition Sd fires. In sub-model $M_{3,i}$, transition $I_{3,i}$ represents the dispatching of events to the completion event handler queue after demultiplexing. The firing rate of transition $I_{3,i}$ is $\eta_{3,i}$ and is given by Equation (2). An inhibitor arc from place CQ_i to transition $I_{3,i}$ with multiplicity C_i prevents the firing

Table 2: Reward rates for performance measures

Sub-model	Perf. Measure	Reward rate
$M_{1,i}$	**Loss probability (L_i)**	return($\#S_i == N_i ? 1 : 0$)
	Busy handlers (B_i)	return($\#S_i$)
	Processing rate ($\alpha_{1,i}$)	return rate(Sr_i)
M_2	**Queue length, common queue (Q)**	return ($\#CQ$)
	Total demultiplexing time (τ_2)	return (($\#CQ + \#DQ)/\kappa$)
	Effective demux. rate (δ)	return rate(Sd)
$M_{3,i}$	**Throughput (T_i)**	return rate(Sc_i)
	Queue length, separate queue (Q_i)	return($\#CQ_i$)
	Total comp. hand. time ($\tau_{3,i}$)	return (($\#CQ_i + \#E_i)/\gamma_i$)

of transition $I_{3,i}$ when the separate completion event queue is full. We note that the capacities of the queues need to be very large to prevent request loss and enable model decomposition. The inhibitor arcs need to be added to the sub-models however, to prevent an overflow of tokens while solving the sub-models using numerical techniques.

The rationale used to assign reward rates to obtain the performance measures from the sub-models is as follows. For sub-model $M_{1,i}$, the busy event handlers B_i is given by the number of tokens in place S_i, the loss probabilility L_i is given by the probability of N_i tokens in place S_i and the processing rate $\alpha_{1,i}$ is given by the firing rate of transition Sr_i. For sub-model M_2, the queue length of the common completion queue Q is the number of tokens in place CQ and the effective demultiplexing rate δ is given by the firing rate of transition Sd. The total demultiplexing time τ_2, according to Little's law [29], is given by the ratio of the sum of the number of tokens in places CQ and DQ and the demultiplexing rate κ. For sub-model $M_{3,i}$, the queue length of the separate completion queue Q_i is given by the number of tokens in place CQ_i, the throughput T_i is the firing rate of transition Sc_i and the total completion event handling time $\tau_{3,i}$ is given by the ratio of the sum of the number of tokens in places CQ_i and E_i and the completion event handling rate γ_i. The total completion event handling time is obtained using Little's law, similar to the total demultiplexing time. These reward rates are summarized in Table 2. In the table the notation $\#$ is used to denote the number of tokens in a place, for example $\#CQ_i$ denotes the number of tokens in place CQ_i. The measures that are indicative of Web server performance (defined in Section 4.2) are in boldface, whereas the ones which facilitate model decomposition are in plain text.

Table 3: Nominal parameter values

Parameter	Value
Arrival rate (λ_i)	10.0/s
Pool size (N_i)	8
Service rate (μ_i)	2.0/s
Capacity, common queue (C)	3000
Demultiplexing rate (κ)	$25.0/s$
Capacity, separate queue (C_i)	3000
Completion event handling rate (γ_i)	25.0/s

5 Illustrative examples

In this section we illustrate the potential of the methodology to guide configuration and provisioning decisions with examples. We consider a Web server which provides two types of services, or $m = 2$. In the first experiment, we validate the performance estimates obtained from the SRN model using simulation. We then conduct several experiments to analyze the impact of the parameters of each one of three sub-models on the performance metrics. In all the experiments, the performance estimates were obtained by solving the SRN sub-models shown in Figure 3 using SPNP. Because the arrival and service rates, and the configuration parameters for both types of requests are set to the same values in all the experiments, it results in nearly similar performance estimates for both request types. As a result, performance estimates of only one request type are reported for all the experiments.

Experiment I: Model validation

The first experiment serves to validate the performance estimates obtained from SRN using simulation, for nominal parameter values summarized in Table 3. For this purpose, the queuing model of the Proactor-based Web server was implemented using CSIM [26]. The performance estimates obtained from SRN and simulation are shown in Table 4. The confidence intervals for the estimates obtained using simulation are within 5% of the mean and are not shown here. The results reported in the table indicate that the performance estimates obtained using simulation match very well with estimates obtained from the model decomposition strategy.

Experiment II: Impact of event handler pool size

Table 4: Comparison of performance measures

Perf. Measure	SRN	Sim.
Loss probability (L_i)	0.07	0.70
Busy handlers (B_i)	4.64	4.65
Queue length, common queue (Q)	2.12	2.13
Queue length, separate queue (Q_i)	0.22	0.24
Response time (R_i)	0.64	0.63
Throughput (T_i)	9.29	9.27

The second experiment analyzes the impact of the sizes of the event handler pools. We consider three request arrival rates, namely, $\lambda_i = 10.0$/s, $\lambda_i = 15.0$/s and $\lambda_i = 20.0$/s. The sizes of the event handler pools were varied from 5 to 25 in steps of 5 and the service rate of a single event handler was set to 2.0/s. The loss probability, and the number of busy event handlers as a function of the event handler pool size are shown in the left and the right plots in Figure 4. These figures indicate that a pool size of 5 is not sufficient to handle the lowest level of load, which leads to a high loss probability and causes the throughput to be lower than the arrival rate. As the pool size increases, the performance improves. However, for a given arrival rate increasing the pool size beyond a certain threshold offers diminishing returns. For example, when the arrival rates are 20.0/s, the loss probability when the pool size is 20 is less than 1%. Thus, increasing the pool size beyond 20 is perhaps not cost-effective, especially, since the rule of thumb suggests that no more than two event handlers be placed on a single processor [23, 22, 14]. The average processing rates of sub-model #1 are lower than the arrival rates when the loss probabilities are greater than 0.0. However, with a pool size of 20 when the loss probabilities are negligible, the average processing rates are the same as the arrival rates.

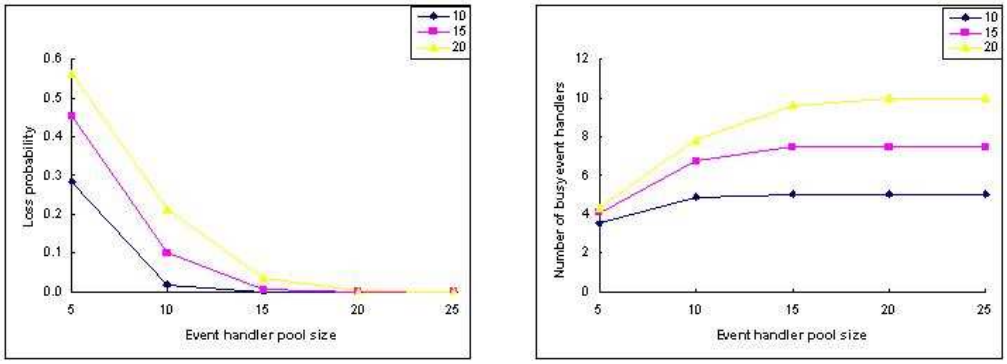

Figure 4: Performance metrics as a function of event handler pool size

Experiment III: Impact of service rate of asynchronous operations

In the third experiment we analyze the impact of service rate of asynchronous operations on the loss probability and the number of busy event handlers. For this purpose, we set the pool size to 10, and vary the service rate of each event handler from 2.0/s to 5.0/s in steps of 0.5/s. The loss probability and busy event handlers for each one of the three arrival rates considered in Experiment II as a function of the event handler service rate are plotted in Figure 5. As intuitively expected, for a given arrival rate the loss probabilities decrease as the event handler service rate increases. Further, the event handler rate for which the loss probability becomes negligible decreases with the arrival rate, which is also expected.

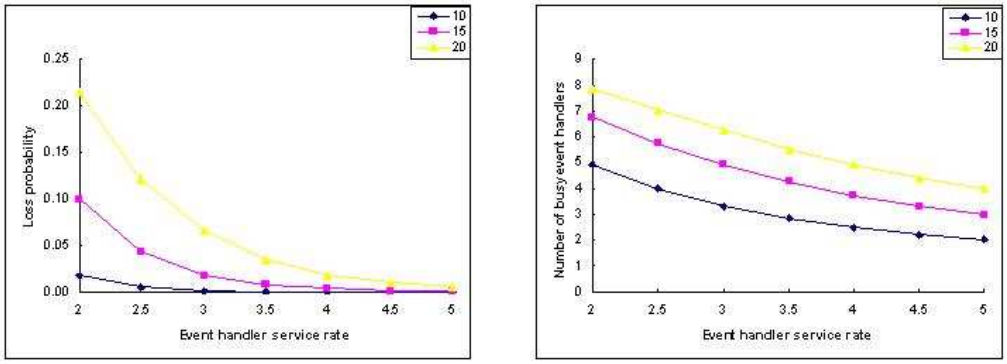

Figure 5: Performance metrics as a function of event handler service rate

The results of Experiments II and III indicate that the goal of negligible loss probability can be achieved in two ways. In the first approach, the size of the event handler pool can be increased for a fixed event handler rate, while in the second approach the event handler rate can be increased for a fixed size of the event handler pool. Typically, an increase in the pool size also requires procurement of additional processors to support the desired level of concurrency, since the rule of thumb as discussed in Experiment II is two threads per processor [23, 22, 14]. On the other hand, it may be possible to increase the event handler service rate to some extent by implementing changes that do not require expensive changes to the hardware infrastructure. Such a change, for example, might include replacing the existing version of the algorithm by a more efficient one.

Experiment IV: Impact of demultiplexing rate

This experiment seeks to analyze the impact of the demultiplexing rate of the Proactor. The queue size of the common completion queue was set to 3000. The sizes of the event handler pools are set to 20 to ensure negligible loss probabilities. The input rates to the completion events to the second sub-model are given by the average processing rates $\alpha_{1,i}$s of the first sub-model. Since the loss probability is negligible, these

input rates are the same as the original arrival rates. The effective input rate seen by the demultiplexer is thus the sum of the average processing rates of the two request types and is given by 20.0/s, 30.0/s, and 40.0/s. For the demultiplexer to maintain the processing rate at the same level as the input rate for all the three input rates, the demultiplexing rate κ should be greater than 40.0/s. κ was thus varied from 40.5/s to 45.0/s, and the two performance metrics, namely, queue length and the contribution of the second sub-model to the response time, namely, τ_2 were obtained. The queue length of the completion queue as a function of the demultiplexing rate is shown in Figure 6. The figure indicates that even for the highest effective input rate ($\lambda_i = 20.0$/s) and the lowest demultiplexing rate the average queue length is about 35. This suggests that a buffer space of 3000, which was used in this experiment, may be unnecessary. Estimates of τ_2 are used in conjunction with the estimates of $\tau_{3,i}$ to obtain the end-to-end response times of the requests.

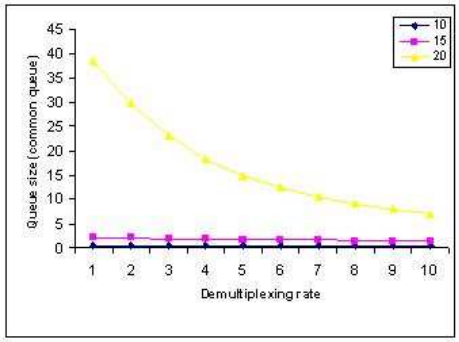

Figure 6: Queue length (common queue) as a function of demultiplexing rate

Experiment V: Impact of completion event handler rate

The input rate the third sub-models were obtained by computing the effective demultiplex rate using Equation (2). The effective demultiplexing rate was obtained when the event handler pool size was set to 20 and the demultiplexing rate was set to 45.0/s. Since these parameter settings ensure that the probability of request loss in sub-models $M_{1,i}$ and M_2 is negligible, the input rates of the completion events to model $M_{3,i}$ are the the same as the original arrival rates. Thus, to ensure a stable queue, the service rates of the completion event handlers should be greater than 20.0/s. Thus, the service rates were varied from 20.5/s to 25.0/s in steps of 0.5/s. The sizes of the separate completion queues were set to 3000. The queue lengths of the separate completion queues as a function of the completion event handler rate is shown in Figure 7. Similar to Experiment IV, the highest average queue length in this case is approximately 39, indicating that provisioning a buffer of 3000 may be unnecessary. For all the completion event handler rates considered, the throughput of the Web server for each request type is identical to the request arrival rate.

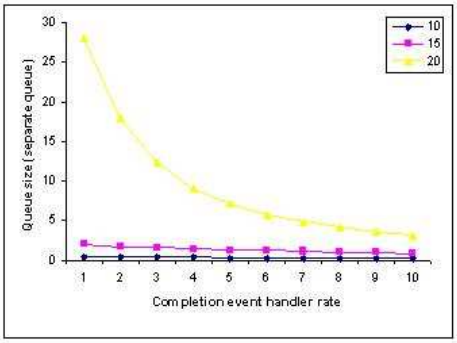

Figure 7: Queue length (separate queue) as a function of completion event handler rate

The estimates of τ_2 obtained from Experiment IV and those of $\tau_{3,r}$ and $\tau_{3,w}$ from Experiment V were combined to obtain the response time as a function of the demultiplexing and completion event handler rates for each arrival rate using Equation (3). Table 5 shows the response times when the request arrival rates are 15.0/s for the purpose of illustration. The response times in Table 5 indicate that for a given value of completion handler rate, increasing the demultiplexing rate from 41.0/s to 45.0/s, provides only a small improvement in the performance. On the other hand, for a given demultiplexing rate, improving the completion handler rate yields better performance benefits. The table also provides opportunities for tradeoffs, by identifying multiple combinations of the completion handler and demultiplexing rates which provide the same response time. For example, the response time is 0.635s for two combinations: (i) handler rate of 24.0/s and demultiplexing rate of 41.0/s, and (ii) handler rate of 23.0/s and demultiplexing rate of 43.0/s. Similarly, the combinations where completion event handler rate is 25.0/s and demultiplexing rate is 43.0/s and completion event handler rate of 24.0/s and demultiplexing rate of 45.0/s provide the same response time of 0.614s. By identifying multiple possible combinations with the same performance, a service provider can choose a combination which is more cost-effective to achieve the same end-to-end response time.

6 Related research

Research efforts in two areas, namely, performance analysis of middleware services and patterns and performance analysis of Web servers, are relevant to the present work.

Performance analysis of middleware services and patterns can be broadly classified into two categories; namely, measurement-based and model-based. The measurement-based approach comprises of testing specific implementations with benchmarking suite(s) and then measuring the relevant metrics [4, 11, 18, 30]. The model-based approach consists of building and solving a model using analytical/numerical or simulation

Table 5: Response times (s), $\lambda_i = 15.0$/s

Completion	Demultiplexing rate (/s)				
handler rate (/s)	41.0	42.0	43.0	44.0	45.0
21.0	0.687	0.680	0.674	0.669	0.664
22.0	0.664	0.657	0.651	0.646	0.641
23.0	0.648	0.641	0.635	0.630	0.626
24.0	0.635	0.629	0.623	0.618	0.614
25.0	0.627	0.620	0.614	0.609	0.604

methods to obtain performance estimates. Ramani *et al.* [21] present a framework for performability analysis of messaging systems in middleware. Aldred *et al.* [1] develop Colored Petri Net (CPN) models for different types of coupling between the application components and with the underlying middleware. Kahkipuro [12] propose a multi-layer performance modeling framework based on UML and queuing networks for CORBA-based systems. The methodology, however, is for generic CORBA-based client/server systems rather than for systems built using design patterns.

With the growing complexity of software systems and increasing pressure to reduce the time to market, there is a significant push towards composing large systems using reusable building blocks or patterns [3, 25]. Performance analysis of such a composed system requires models of the individual building blocks and their composition. In this paper we have developed a performance model of the Proactor pattern. Although the model was developed to analyze the performance of an asynchronous Web server, it is generic and could be used for the performance analysis of any Proactor-based system. Our previous work has developed an analytical model of the Reactor pattern [5].

Web server performance analysis can also be broadly classified into measurement-based and model-based approaches. The former approach measures the server performance using benchmarks [9, 8, 10]. Many model-based approaches use queuing networks to analyze performance [27, 6, 2, 17, 28, 13]. However, most of these techniques are for Web servers that use synchronous mechanisms for concurrency, whereas the model presented in this paper is applicable for the analysis of an asynchronous Web server.

7 Conclusions and future research

In this paper we presented a model-based approach for the design-time performance analysis of a Web server which implements concurrent processing capabilities using the asynchronous mechanisms encapsulated in

the Proactor pattern. We represented the characteristics of the Proactor pattern that are relevant from a performance perspective in the form of a queuing model. We then presented a model decomposition strategy to enable the application of the model in practical scenarios. We illustrated the use of the model to guide configuration and provisioning decisions with several examples. Our future research consists of developing an analytical/numerical approach for the performance analysis of the Proactor pattern. Developing performance models for other patterns such as the Active object is also a topic of future research.

Acknowledgments

This research was supported by the following grants from the National Science Foundation (NSF): Univ. of Connecticut (CNS-0406376 and CNS-SMA-0509271), Vanderbilt Univ. (CNS-SMA-0509296) and Univ. of Alabama at Birmingham (CNS-SMA-0509342).

References

[1] L. Aldred, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. "On the notion of coupling in communication middleware". In *Proc. of Intl. Symposium on Distributed Objects and Applications (DOA)*, pages 1015–1033, Agia Napa, Cyprus, 2005.

[2] J. Cao, M. Andersson, C. Nyberg, and M. Kihl. Web server performance modeling using an M/G/1/K*PS queue. In *10th International Conference on Telecommunications (ICT'03)*, pages 1501–1506, 2003.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[4] A. Gokhale and D. C. Schmidt. "Measuring and optimizing CORBA latency and scalability over high-speed networks". *IEEE Trans. on Computers*, 47(4), April 1998.

[5] S. Gokhale, A. Gokhale, and J. Gray. "Response time analysis of an event demultiplexing pattern in middleware for network services". In *Proc. of IEEE Global Telecommunications Conference (GLOBECOM), Symposium on Advances for Networks and Internet*, St. Louis, MO, November 2005.

[6] J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of HTTP over several transport protocols. *IEEE/ACM Transactions on Networking*, 5(5):616–630, 1997.

[7] C. Hirel, B. Tuffin, and K. S. Trivedi. "SPNP: Stochastic Petri Nets. Version 6.0". *Lecture Notes in Computer Science 1786*, 2000.

[8] J. C. Hu, I. Pyarali, and D. C. Schmidt. “Measuring the impact of event dispatching and concurrency models on Web server performance over high-speed networks”. In *Proc. of GLOBECOM*, pages 1024–1031, 1997.

[9] Y. Hu, A. Nanda, and Q. Yang. Measurement, analysis and performance improvement of the Apache Web server. In *IEEE International Performance, Computing and Communications Conference (IPCCC’99)*, pages 261–267, 1999.

[10] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High-performance Web site design techniques. *IEEE Internet Computing*, 4(2):17–26, March 2000.

[11] M. Juric, I. Rozman, M. Hericko, and T. Domajnko. “CORBA, RMI and RMI-IIOP performance analysis and optimization”. In *Proc. of SCI 2000*, pages 582–587, Orlando, FL, July 2000.

[12] P. Kahkipuro. *“Performance modeling framework for CORBA based distrbuted systems”*. PhD thesis, Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland, May 2000.

[13] K. Kant and C. R. M. Sundaram. A server performance model for static Web workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’00)*, pages 201–206, 2000.

[14] Y. Ling, T. Mullen, and X. Lin. Analysis of. optimal thread pool size. *ACM SIGOPS Operating System Review*, 34(2):42–55, 2000.

[15] Apache Software Foundation. Apache HTTP server project. `http://httpd.apache.org/`.

[16] D. Menascé. Web server software architecture. *IEEE Internet Computing*, 7(6):78–81, 2003.

[17] R. Nossenson and H. Attiya. The N-burst/G/1 model with heavy-tailed service-times distribution. In *Proceedings of 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS’04)*, pages 131–138, 2004.

[18] O. Othman, J. Balasubramanian, and D. C. Schmidt. “Performance evaluation of an adaptive middleware load balancing and monitoring service”. In *Proc. of the 24th IEEE Intl. Conference on Distributed Computing Systems*, pages 135–146, Tokyo, Japan, May 2004.

[19] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[20] A. Puliafito, M. Telek, and K. S. Trivedi. “The evolution of stochastic Petri nets”. In *Proc. of World Congress on Systems Simulation*, pages 3–15, Singapore, September 1997.

[21] S. Ramani, K. Goseva-Popstojanova, and K. S. Trivedi. “A framework for performability modeling of messaging services in distributed systems”. In *Proc. of 8th IEEE Intl. Conference on Engineering of Complex Computer Systems (ICECCS 02)*, Greenbelt, MD, December 2002.

[22] J. Richter. *Advanced Windows (3rd Edition)*. Microsoft Press, 1996.

[23] J. Richter. *Programming Server-Side Applications for Microsoft Windows 2000*. Microsoft Press, 2000.

[24] R. A. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, Boston, 1996.

[25] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[26] H. Schwetman. “CSIM reference manual (revision 16)”. Technical Report ACA-ST-252-87, Microelectronics and Computer Technology Corp., Austin, TX.

[27] L. Slothouber. A model of Web server performance. In *Proceedings of the Fifth International World Wide Web Conference*, 1996.

[28] M. S. Squillante, D. D. Yao, and L. Zhang. Web traffic modeling and Web server performance analysis. In *Proceedings of the 38th Conference on Decision and Control*, pages 4432–4439, 1999.

[29] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley, 2001.

[30] P. Tuma and A. Buble. “Overview of the CORBA performance”. In *Proc. of the 2002 EurOpen CZ Conference*, September 2002.

[31] R. D. van der Mei, R. Hariharan, and P. Reeser. Web server performance modeling. *Telecommunication Systems*, 16(3-4):361–378, 2001.