Performance Analysis of the Reactor Pattern in Network Services

Swapna Gokhale	Aniruddha Gokhale	Jeff Gray
Paul Vandal		
Upsorn Praphamontripong		
Dept. of CSE	Dept. of EECS	Dept. of CIS
University of Connecticut	Vanderbilt University	U of Alabama, Birmingham
Storrs, CT	Nashville, TN	Birmingham, AL
ssg@engr.uconn.edu	a.gokhale@vanderbilt.edu	gray@cis.uab.edu

Abstract

The growing reliance on services provided by software applications places a high premium on the reliable and efficient operation of these applications. A number of these applications follow the event-driven software architecture style since this style fosters evolvability by separating event handling from event demultiplexing and dispatching functionality. The event demultiplexing capability, which appears repeatedly across a class of event-driven applications, can be codified into a reusable pattern, such as the Reactor pattern. In order to enable performance analysis of event-driven applications at design time, a model is needed that represents the event demultiplexing and handling functionality that lies at the heart of these applications. In this paper, we present a model of the Reactor pattern based on the well-established Stochastic Reward Net (SRN) modeling paradigm. We discuss how the model can be used to obtain several performance measures such as the throughput, loss probability, and upper and lower bounds on the response time. We illustrate how the model can be used to obtain the performance metrics of a Virtual Private Network (VPN) service provided by a Virtual Router (VR). We validate the estimates of the performance measures obtained from the SRN model using simulation.

1 Introduction

Service oriented computing (SoC), which is made feasible by middleware-based distributed systems, is an emerging technology to provide the next-generation services to meet societal needs ranging from basic necessities, such as education, energy, communications and healthcare to emergency and disaster management. For SOC to be successful in meeting the demands of society, assurance on the performance of these services is necessary. Since these services are primarily built using communication middleware, the problem reduces to the issue of performance assurance of the middleware platforms.

Middleware typically comprises a number of building blocks, which are essentially patterns-based reusable software frameworks. The building blocks are then combined in a variety of ways to provide a complete middleware solution for hosting the services. However, the middleware problem is made more complex due to the large number of options available for each building block of the middleware. The choice of building blocks and their configuration options have an impact on the performance of the services provided by the systems.

The current state of the art in middleware performance analysis requires configuring, integrating and composing the building blocks to form entire middleware solutions, which are then evaluated via empirical benchmarking and extensive profiling. Any ill desired effects of the choices made in the configuration, composition and integration can be determined only very late in the lifecycle, which can be detrimental to system development costs and schedules. In order to enable the right design choices, a systematic methodology to analyze the performance of these systems at design time is necessary. Such a methodology may consist of models to analyze the performance of individual building blocks comprising the middleware and the composition of these building blocks.

The performance models may be based upon well-known analytical/numerical modeling paradigms [14, 1, 5] and simulation techniques [17, 19]. As a first step towards the development of such a methodology, this paper presents a model of the Reactor pattern [2, 16], which provides important synchronous demultiplexing and dispatching capabilities to network services and applications. The model is based on the Stochastic Reward Net (SRN) modeling paradigm [14]. Our previous efforts [3] have discussed the use of the model to obtain an upper bound on the response time of a service. In this paper we extend our previous work and describe how the model can be used to obtain several additional performance measures including throughput and loss probability. We also demonstrate how the model can be used to obtain the lower bound on the response time. We illustrate how the model can be used to estimate the performance metrics of a Virtual Private Network (VPN) service provided by a Virtual Router (VR) [9]. We validate the performance metrics obtained from the model using simulation.

Paper organization: The paper is organized as follows: Section 2 presents the performance model of the Reactor pattern. Section 3 illustrates how the performance model of the Reactor pattern can be used to estimate the performance metrics of a VPN service provided by a VR. Section 4 offers concluding remarks and directions for future research.

2 Performance Model of the Reactor Pattern

This section provides a context for understanding the contribution of this paper summarizing our earlier work. We first provide an overview of the Reactor followed by the SRN model of the Reactor pattern.

2.1 Reactor Pattern in Middleware Implementations

Figure 1 depicts a typical event demultiplexing and dispatching mechanism documented in the Reactor pattern. The application registers an event handler with the event demultiplexer and delegates to it the responsibility of listening for incoming events. On the occurrence of an event, the demultiplexer dispatches the event by making a callback to its associated application-supplied event handler. This is the idea behind the Reactor pattern, which provides synchronous event demultiplexing and dispatching capabilities.

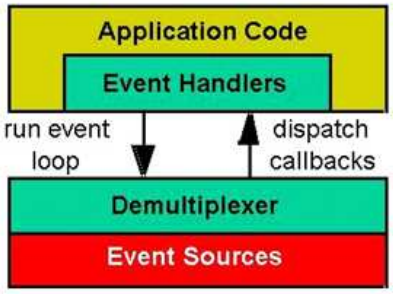

Figure 1. Event Demultiplexers in Middleware

The Reactor pattern can be implemented in many different ways depending on the event demultiplexing capabilities provided by the underlying operating system and the concurrency requirements of the applications. For example, the demultiplexing capabilities of a Reactor could be based on the `select ()` or `poll ()` system calls provided by POSIX-compliant operating systems, or `WaitForMultipleObject ()` as found in the different flavors of Win32 operating systems. Moreover, the handling of the event in the event handler could

be managed by the same thread of control that was listening for events leading to a single-threaded Reactor implementation. Alternatively, the event could be delegated to a pool of threads to handle the events leading to a thread-pool Reactor.

2.2 Characteristics of the Reactor Pattern

We consider a single-threaded, *select*-based implementation of the Reactor pattern with the following characteristics:

- The Reactor receives two types of input events with one event handler for each type of event registered with the Reactor.
- Each event type has a separate queue, which holds the incoming events of that type. The buffer capacity for the queue of type #1 events is denoted N_1 and of type #2 events is denoted N_2.
- Event arrivals for both types of events follow a Poisson process with rates λ_1 and λ_2, while the service times of the events are exponentially distributed with rates μ_1 and μ_2.
- In a snapshot, an event of type #1 is serviced with a higher priority over an event of type #2. In other words, when event handles corresponding to both event types are enabled in a snapshot, the event handle corresponding to type #1 is serviced with a priority that is higher than the event handle of type #2.

2.3 Desired Performance Metrics

The following performance metrics are of interest for each one of the event types in the reactor pattern described in Section 2.2:

- **Expected throughput** – which provides an estimate of the number of events that can be processed by the single threaded event demultiplexing framework. These estimates are important for many applications, such as telecommunications call processing.
- **Expected queue length** – which provides an estimate of the queuing for each of the event handler queues. These estimates are important to develop appropriate scheduling policies for applications with real-time requirements.

- **Probability of event loss** – which indicates how many events will have to be discarded due to lack of buffer space. These estimates are important particularly for safety-critical systems, which cannot afford to lose events. These also provide an estimate on the desired levels of resource provisioning.

- **Expected response time** – which indicates the time taken to service an event. It is also important to establish lower and upper bounds on the response time. These estimates, especially, the upper bound can allow us to determine whether the deadlines for real-time services can be satisfied in the worse case.

2.4 SRN Model

For completeness sake we reproduce the SRN model of the Reactor pattern illustrated in our earlier work [3]. A Stochastic Reward Net (SRN) substantially extends the modeling power of Generalized Stochastic Petri Nets (GSPNs) [14], which are an extension of Petri nets [13]. A SRN is a modeling technique that is concise in its specification and closer to a designer's intuition about what a model should look like. SRNs have been extensively used for performance, reliability and performability analysis of a variety of systems [15, 6, 7, 18, 8, 11]. The work closest to the proposed research is reported by Ramani *et al.* [15], where SRNs are used for the performance analysis of the CORBA event service. A detailed overview of SRNs can be obtained from [14].

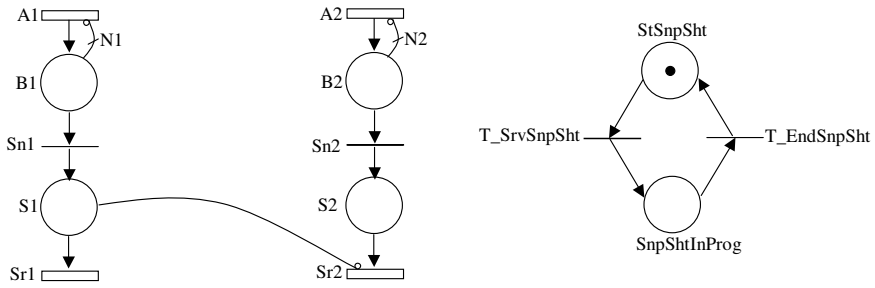

Figure 2. SRN model for the Reactor pattern

Figure 2 shows the SRN model for the Reactor pattern with the characteristics described in Section 2.2. Table 1 summarizes the enabling/guard functions for the transitions in the net. The net on the left-hand side models the arrival, queuing and service of the two types of events. Transitions $A1$ and $A2$ represent the arrival of the events of type #1 and #2, respectively. Places $B1$ and $B2$ represent the queue for the two types of events. Transitions

$Sn1$ and $Sn2$ are immediate transitions that are enabled when a snapshot is taken. Places $S1$ and $S2$ represent the enabled handles of the two types of events, whereas transitions $Sr1$ and $Sr2$ represent the execution of the enabled event handlers of the two types of events. An inhibitor arc from place $B1$ to transition $A1$ with multiplicity $N1$ prevents the firing of transition $A1$ when there are $N1$ tokens in place $B1$. The presence of $N1$ tokens in place $B1$ indicates that the buffer space to hold the incoming input events of the first type is full, and no additional incoming events can be accepted. The inhibitor arc from place $B2$ to transition $A2$ achieves the same purpose for type #2 events. The inhibitor arc from place $S1$ to transition $Sr2$ prevents the firing of transition $Sr2$ when there is a token in place $S1$. This models the prioritized service for an event of type #1 over event of type #2 in a given snapshot.

The net on the right of Figure 2 models the process of taking successive snapshots and prioritized service of the event handle corresponding to type #1 events in each snapshot. Transition $Sn1$ is enabled when there is a token in place $StSnpSht$, at least one token in place $B1$, and no tokens in place $S1$. Similarly, transition $Sn2$ is enabled when there is a token in place $StSnpSht$, at least one token in place $B2$, and no tokens in place $S2$. Transition $T\_SrvSnpSht$ is enabled when there is a token in either one of the places $S1$ and $S2$, and the firing of this transition deposits a token in place $SnpShtInProg$.

The presence of a token in the place $SnpShtInProg$ indicates that the event handles that were enabled in the current snapshot are being serviced. After these event handles complete execution, the current snapshot is complete and it is time to take another snapshot. This is accomplished by enabling the transition $T\_EndSnpSht$. Transition $T\_EndSnpSht$ is enabled when there are no tokens in both places $S1$ and $S2$. Firing of the transition $T\_EndSnpSht$ deposits a token in place $StSnpSht$, indicating that the service of the enabled handles in the present snapshot is complete, which marks the initiation of the next snapshot.

Table 1. Enabling/Guard functions

Transition	Guard function
$Sn1$	$((\#StSnpShot == 1)\&\&(\#B1 >= 1)\&\&(\#S1 == 0))?1 : 0$
$Sn2$	$((\#StSnpShot == 1)\&\&(\#B2 >= 1)\&\&(\#S2 == 0))?1 : 0$
$T\_SrvSnpSht$	$((\#S1 == 1)\|\|(\#S2 == 1))?1 : 0$
$T\_EndSnpSht$	$((\#S1 == 0)\&\&(\#S2 == 0))?1 : 0$

We now describe how the process of taking a single snapshot is modeled by the SRN model presented in Figure 2. We consider a scenario where there is one token in each one of the places $B1$ and $B2$, and there is a token in the place $StSnpSht$. Also, there are no tokens in places $S1$ and $S2$. In this scenario, transitions $Sn1$ and $Sn2$ are enabled. Both these transitions are assigned the same priority, and any one of these transitions can fire

first. Also, since these transitions are immediate, their firing occurs instantaneously. Without loss of generality, it can be assumed that transition $Sn1$ fires before $Sn2$, which deposits a token in place $S1$.

When a token is deposited in place $S1$, transition $T\_SrvSnpSht$ is enabled. In addition, transition $Sn2$ is already enabled. If transition $T\_SrvSnpSht$ were to fire before transition $Sn2$, it would disable transition $Sn2$, and prevent the handle corresponding to the second event type from being enabled. In order to prevent transition $T\_SrvSnpSht$ from firing before transition $Sn2$, transition $T\_SrvSnpSht$ is assigned a lower priority than transition $Sn2$. Because transitions $Sn1$ and $Sn2$ have the same priority, this also implies that the transition $T\_SrvSnpSht$ has a lower priority than transition $Sn1$. This ensures that in a given snapshot, event handles corresponding to each event type are enabled when there is at least one event in the queue.

After both event handles are enabled, transition $T\_SrvSnpSht$ fires and deposits a token in place $SnpShtInProg$. The presence of a token in the place $SnpShtInProg$ indicates that the event handles that were enabled in the current snapshot are being serviced. The event handle corresponding to type #1 event is serviced first, which causes transition $Sr1$ to fire and the removal of the token from place $S1$. Subsequently, transition $Sr2$ fires and the event handle corresponding to the event of type #2 is serviced. This causes the removal of the token from place $S2$. After both events are serviced and there are no tokens in places $S1$ and $S2$, transition $T\_EndSnpSht$ fires, which marks the end of the present snapshot and the beginning of the next one.

The performance measures described in Section 2.3 can be computed by assigning reward rates at the net level as summarized in Table 2. The throughputs of events of type #1 and #2 denoted T_1 and T_2 respectively are given by the rate at which transitions $Sr1$ and $Sr2$ fire. The queue lengths of the events denoted Q_1 and Q_2 are given by the number of tokens in places $B1$ and $B2$ respectively. The total number of events of type #1 denoted E_1 is given by the sum of the number of tokens in places $B1$ and $S1$. Similarly, the total number of events of type #2 denoted E_2 is given by the sum of the number of tokens in places $B2$ and $S2$. The loss probability of type #1 events denoted L_1 is given by the probability of $N1$ tokens in place $B1$. Similarly, the loss probability of type #2 events denoted L_2 is given by the probability of $N2$ tokens in place $B2$.

We obtain the response times of the events using the tagged customer approach [10]. In the tagged customer approach, an arriving event is tagged and its trajectory through the system is followed from entry to exit. The response time of the tagged event is then determined conditional to the state in which the system lies when the event arrives. The unconditional response time can be obtained as the weighted sum of the conditional response times, with the weights given by the steady state probabilities of being in each one of the states. Typically, the

response time of an event consists of two pieces; namely, the time taken to service the event hereafter referred to as the "service time," and the time that the event must wait in the system before its service commences, hereafter referred to as "waiting time". In our case, the average service time of an incoming type #1 and type #2 event is given by $1/\mu_1$ and $1/\mu_2$, irrespective of the state in which the system lies when the event arrives. The waiting time, however, will depend on the system state. Next, we discuss how the conditional waiting time of each event type is determined.

The conditional waiting time for a tagged event of type #1 will depend on the state of the system, where the state is given by the number of tokens or markings of places $S1$, $S2$, $B1$ and $B2$. Of these four places, the markings of the places $S1$ and $S2$ determine the progress of the current snapshot, whereas, the markings of places $B1$ and $B2$ determine the state of the queue. The mean time taken to complete the current snapshot is given by the sum of two terms, the first term is the product of the number of tokens in place $S1$ and $1/\mu_1$, and the second term is the product of the number of tokens in place $S2$ and $1/\mu_2$. Even if there are no additional events in the queues, the current snapshot must be completed before the service of an incoming event of type #1 can begin. Hence, the time taken to complete the current snapshot contributes to the waiting time of the incoming or tagged type #1 event. In order to obtain the entire waiting time of a tagged type #1 event, the contribution of the queued events of type #1 and type #2 needs to be determined.

Let n_1 be the number of events of type #1 in the queue, and n_2 be the number of events of type #2 in the queue, when the tagged event of type #1 arrives. This implies that after n_1 snapshots the tagged event will be serviced. The following three possibilities arise between the relative values of n_1 and n_2. If $n_1 \leq n_2$, then only n_1 of the type #2 events need to be serviced before the service of the tagged type #1 event can commence, and hence the waiting time is given by $n_1(1/\mu_1 + 1/\mu_2)$. If $n_1 = n_2$, then n_1 events of type #1 and type #2 need to be serviced before the service of the incoming type #1 event can commence, and hence the waiting time is given by $n_1(1/\mu_1 + 1/\mu_2)$. If $n_1 > n_2$, then in the optimistic case, n_1 events of type #1 and n_2 events of type #2 need to be serviced before the service of the tagged event can commence. The optimistic case assumes that no additional events of type #2 arrive in the first n_1 snapshots. In the pessimistic case, however, $n_1 - n_2$ events of type #2 will arrive while the first n_2 events are being serviced. Thus, in the optimistic case, the waiting time will be $n_1/\mu_1 + n_2/\mu_2$, and in the pessimistic case, the waiting time will be $n_1(1/\mu_1 + 1/\mu_2)$. The optimistic case provides the lower bound and the pessimistic case provides the upper bound of the response times. The reward rates to obtain the optimistic and the pessimistic response times of the events of type #1 and type #2 are

summarized in Table 2. In the table, $R_{1,o}$ and $R_{1,p}$ denote the optimistic and pessimistic response times of type #1 events, and $R_{2,o}$ and $R_{2,p}$ denote the optimistic and pessimistic response times of type #2 events.

Table 2. Reward assignments to obtain performance measures

Performance metric	Reward rate
T_1	return rate($Sr1$)
T_2	return rate($Sr2$)
Q_1	return ($\#B1$)
Q_2	return ($\#B2$)
L_1	return ($\#B1 == N1?1 : 0$)
L_2	return ($\#B2 == N2?1 : 0$)

$R_{1,o}$	if ($\#B1 < N1$)
	{
	if ($\#B1 \leq \#B2$)
	return($(1/\mu_1 * (\#S1 + \#B1 + 1) + 1/\mu_2 * (\#S2 + \#B1))$)
	else
	return($(1/\mu_1 * (\#S1 + \#B1 + 1) + 1/\mu_2 * (\#S2 + \#B2))$)
	}
	else
	return(0.0)

$R_{1,p}$	if ($\#B1 < N1$)
	return($(1/\mu_1 * (\#S1 + \#B1 + 1) + 1/\mu_2 * (\#S2 + \#B1))$)
	else
	return(0.0)

$R_{2,o}$	if ($\#B2 < N2$)
	{
	if ($\#B1 < \#B2$)
	return($(1/\mu_1 * (\#S1 + \#B1) + 1/\mu_2 * (\#S2 + \#B2 + 1))$)
	else if ($\#B1 == \#B2$)
	return($(1/\mu_1 * (\#S1 + \#B2) + 1/\mu_2 * (\#S2 + \#B2 + 1))$)
	else
	return($(1/\mu_1 * (\#S1 + \#B2 + 1) + 1/\mu_2 * (\#S2 + \#B2 + 1))$)
	}
	else
	return(0.0)

$R_{2,p}$	if ($\#B2 < N2$)
	return($(1/\mu_2 * (\#S2 + \#B2 + 1) + 1/\mu_1 * (\#S1 + \#B2 + 1))$)
	else
	return(0.0)

2.5 Model Variations

In the model of the reactor pattern described above, the arrival, service and failure distributions are assumed to be exponential. For certain types of applications, this assumption may not hold. For example, for safety-critical applications, events may occur at regular intervals, in which case the arrival process is deterministic. In addition to the deterministic distribution, the arrival, service and failure processes may also follow any other non-exponential or general distribution. There are two ways to consider non-exponential distributions in the SRN model. In the first method, a non-exponential distribution can be approximated using a phase-type approximation [14], and the resulting SRN model can then be solved using SPNP [4]. In the second method, the model can be simulated using discrete-event simulation incorporated in SPNP.

2.6 Model Validation

In order to inspire confidence in the estimates of the performance measures produced by the model, these measures must be validated using simulation and experimentation. In this paper we validated the measures using simulation, leaving the experimental validation for future work. The simulation was implemented using CSIM [17].

3 Case Study: VPN Service using Virtual Router

In this section we describe how the SRN model of the reactor pattern presented in Section 2.4 is used to estimate the response time of a Virtual Private Network (VPN) service provided by a Virtual Router (VR).

Figure 3 illustrates the architecture of a provider-provisioned virtual private network (PPVPN) ([12]) using a VR. A VR is a software/hardware component that is part of a physical router called the provider edge (PE) router. A VR contains the mechanisms to provide highly scalable, differentiated levels of services in VPN architectures. Multiple VRs can reside on a PE device. VRs can be arranged in a hierarchical fashion within a single PE as shown in Figure 3. Moreover, an entity acting as a service provider for an end customer might itself be a customer of a larger service provider. VRs may also use different backbones to improve reliability or to provide differentiated levels of service to customers.

Customer edge (CE) devices wishing to join a VPN connect to a VR on the PE device. A VR can multiplex several distinct CEs belonging to the same VPN session. A VR may use tunneling mechanisms to use multiple routing protocols and link layer protocols, such as IPSec, GRE, and IP-in-IP, to connect with the CEs. A totally

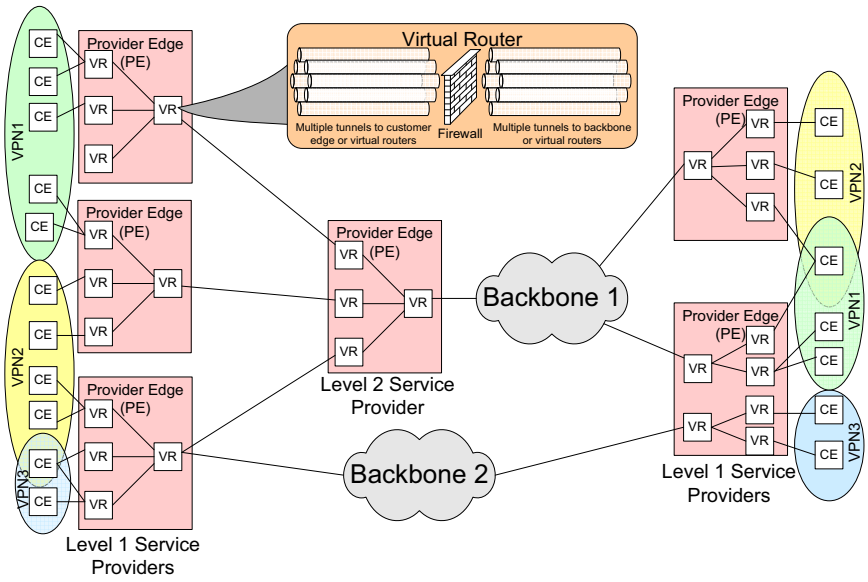

Figure 3. VPN Architecture using Virtual Routers

different set of protocols and tunneling mechanisms could be used for inter-VR or VR-backbone communication. These tunneling mechanisms can also be the basis for differentiated levels of service as well as to provide improved reliability. A VR also comprises firewall capabilities.

We consider a scenario where a VR is used to provide VPN services to two organizations, with each organization having a customer edge (CE) router connected to the VR. The employees of each organization issue VPN set up and tear down service requests to the VR via CEs. Also, the VR offers a differentiated level of service, with organization #1 receiving prioritized service over organization #2. From the point of view of the employees of the organizations, it is necessary that the service requests be handled in a reasonable amount of time. Also, the probability of denying the service requests should be minimal. From the point of view of the VPN service provider, the rate at which the service requests are processed or the throughput is important. The throughput will determine determine the revenue collected by the provider.

In order to implement the VPN service, a reactor pattern with the characteristics described in Section 2.2 can be used to demultiplex the events. The SRN model of the reactor pattern can thus be used for the performance analysis of the VPN service provided by the VR. In order to use the SRN model, we designate the requests originating from organization #1 as events of type #1 and requests originating from organization #2 as events of type #2.

In the early stages of application development lifecycle, it is necessary to analyze the impact of design choices

and configuration parameters on the performance metrics. One of the design choices of the reactor pattern and the VPN service is the buffer space to hold the incoming events of each type. This choice will have a direct impact on all the performance metrics. Most importantly, from the employees' perspective, the buffer space will influence the probability of denying the service requests.

We analyze the impact of the buffer capacities on the performance measures. The values of the remaining parameters (except for the buffer capacities) are reported in Table 3. We consider two values of buffer capacities N_1 and N_2. In the first case, the buffer capacity is set to 1 for both types of events, whereas in the second case the buffer capacity of both types of events is set to 5. The performance metrics for both these cases are summarized in Table 4. Because the values of the parameters of the service requests from organization #1 (λ_1, μ_1 and N_1) are the same as the values of the parameters for the service requests from organization #2 (λ_2, μ_2, and N_2), the throughputs, queue lengths, and the loss probabilities are the same for both types of service requests for each one of the buffer capacities as indicated in Table 4. It can be observed that the loss probabilities are significantly higher when the buffer capacity is 1 compared to the case when the buffer space is 5. Also, due to the higher loss probability, the throughput is slightly lower when the buffer capacity is 1 than when the buffer capacity is 5.

Table 4 also summarizes the estimates of the performance measures obtained using simulation. As indicated in the table, the estimates of the performance measures obtained using the model match with the estimates obtained using simulation for both buffer capacities. Further, the average response times of the events obtained using simulation lie between the upper and lower bounds obtained from the model.

Table 3. Parameter values

Event type	Arrival rate	Service rate
#1	λ_1 = 0.400/sec.	μ_1 = 2.000/sec.
#2	λ_2 = 0.400/sec.	μ_2 = 2.000/sec.

In the early stages, it is rarely the case that the values of the input parameters can be estimated with certainty, which makes it imperative to analyze the sensitivity of the performance metrics to the variations in the input parameters. Sensitivity analysis will enable the provider to determine the regions of operation during which service performance is acceptable. The SRN model can be easily used to assess the sensitivity of the performance measures to the variations in the arrival and service rates. For the sake of illustration, we determine the variations in the response times as a function of the arrival rates λ_1 and λ_2. Towards this end, we vary the arrival rates of the events from 0.4/sec to 1.8/sec one at a time. The highest setting of the arrival rate was chosen to be 1.8/sec.

Table 4. Impact of buffer capacity on performance measures

Performance measure	Buffer space			
	$N_1 = 1, N_2 = 1$		$N_1 = 5, N_2 = 5$	
	SRN	Simulation	SRN	Simulation
T_1	0.37/sec.	0.37/sec.	0.39/sec.	0.39/sec.
T_2	0.37/sec.	0.37/sec.	0.39/sec.	0.39/sec.
Q_1	0.06	0.06	0.12	0.12/sec.
Q_2	0.06	0.06	0.12	0.12/sec.
L_1	0.06	0.06	0.00024	0.00026
L_2	0.06	0.06	0.00024	0.00026
$R_{1,o}$	0.63 sec.	0.67 sec.	0.79 sec.	0.80 sec.
$R_{1,p}$	0.63 sec.		0.83 sec.	
$R_{2,o}$	0.65 sec.	0.72 sec.	0.82 sec.	0.86 sec.
$R_{2,p}$	1.10 sec.		1.33 sec.	

to ensure that the arrival rate of an event is always less than the corresponding service rate of the event. If there is a possibility that the arrival rate exceeds the service rate, then a redesign to ensure higher service rate may be necessary. For each setting of the arrival rate, we obtain the lower and the upper bounds of the response times from the SRN model. We also obtain an estimate of the average response time using simulation.

Figure 4 show the upper and lower bounds of the response time as well as the expected response time for events of type #1 and type #2 as a function of event arrival rate λ_1. Referring to the left plot in the figure, it can be observed that when the arrival rate λ_1 is low, the response time of the events of type #1 approaches the lower bound. As the arrival rate increases, the response time approaches the upper bound. The right plot in Figure 4 indicates that the lower and the upper bounds of the response time of type #2 events are not very different from each other for the entire range of variation of λ_1. Also, the average response time obtained from the simulation are within the two bounds. The response time (lower and upper bounds, average) of events of type #1 and type #2 as a function of λ_2 follow the same trends and are not shown here due to space limitations. The plots also establish that the average response time estimated using simulation always lies between the bounds of the response times obtained from the model for the entire range of variation of λ_1 and λ_2. The model can be used to obtain the lower and the upper bounds of the response times with high confidence. This can facilitate an exploration of the design space beyond what would otherwise be permitted using cumbersome and lengthy simulations.

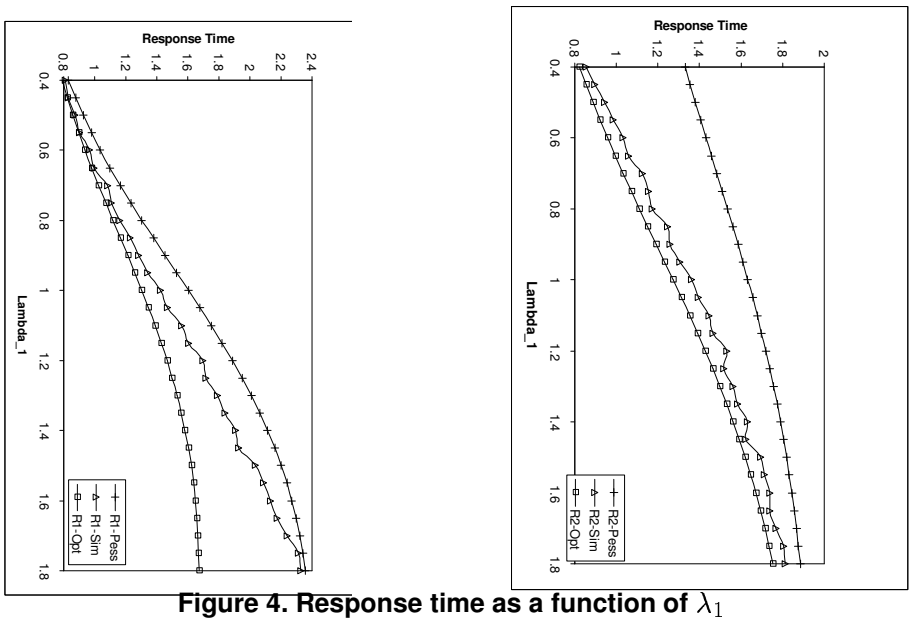

Figure 4. Response time as a function of λ_1

4 Conclusions and future research

In this paper we presented a performance model of the Reactor pattern which offers the important synchronous demultiplexing and dispatching capabilities in middleware. The model was based on the Stochastic Reward Net (SRN) modeling paradigm. We illustrated how the performance model could be used to obtain an estimate of the response time of a VPN service provided by a Virtual Router (VR). Our future research consists of empirically validating the response time estimates obtained from the performance model. Developing and validating the performance models of other middleware building blocks and the composition of these building blocks 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] H. Choi, V. Kulkarni, and K. S. Trivedi. Markov Regenerative Stochastic Petri Net. *Performance Evaluation*, 20(1–3):337–357, 1994.

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

[3] S. Gokhale, A. S. Gokhale, and J. Gray. Response time analysis of middleware event demultiplexing pattern for network services. In *Proc. of IEEE Globecom, Advances for Networks and Internet Track*, St. Louis, MO, December 2005.

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

[5] G. Horton, V. Kulkarni, D. Nicol, and K. S. Trivedi. Fluid stochastic Petri nets: Theory, application and solution techniques. *Journal of Operations Research*, 405, 1998.

[6] O. Ibe, A. Sathaye, R. Howe, and K. S. Trivedi. Stochastic Petri net modeling of VAXCluster availability. In *Proc. of Third International Workshop on Petri Nets and Performance Models*, pages 142–151, Kyoto, Japan, 1989.

[7] O. Ibe and K. S. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, December 1990.

[8] O. Ibe and K. S. Trivedi. Stochastic Petri net analysis of finite–population queueing systems. *Queueing Systems: Theory and Applications*, 8(2):111–128, 1991.

[9] P. Knight, H. Ould-Brahim, and B. Gleeson. Network based VPN Architecture using Virtual Routers. *IETF Network Working Group Internet Draft, draft-ietf-l3vpn-vpn-vr-02.txt*, pages 1–21, Apr. 2004.

[10] B. Melamed and M. Yadin. Randomization procedures in the computation of cumulative-timed distributions over discrete-state markov process. *Operations Research*, 32(4):926–944, July-August 1984.

[11] J. Muppala, G. Ciardo, and K. S. Trivedi. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintainability and Serviceability: An International Journal Published by SAE Internationa*, 1(2):9–20, July 1994.

[12] A. Nagarajan. Generic Requirements for Provider Provisioned Virtual Private Network (PPVPN). In A. Nagarajan, editor, *IETF Network Working Group Request for Comments, RFC 3809*, pages 1–25. IETF, June 2004.

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

[14] 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.

[15] S. Ramani, K. S. Trivedi, and B. Dasarathy. Performance analysis of the CORBA event service using stochastic reward nets. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 238–247, October 2000.

[16] 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.

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

[18] H. Sun, X. Zang, and K. S. Trivedi. A stochastic reward net model for performance analysis of prioritized DQDB MAN. *Computer Communications, Elsevier Science*, 22(9):858–870, June 1999.

[19] The VINT Project. Network Simulator - NS-2. `http://www.isi.edu/nsnam/ns`, 1996.