Performance Analysis of Middleware Event Demultiplexing Patterns in Distributed High Performance Software Systems

Swapna S. Gokhale
Dept. of Computer Science
and Engineering
University of Connecticut
Storrs, CT
{ssg@cse.uconn.edu}

Aniruddha Gokhale
Dept. of Electrical Engineering
and Computer Science
Vanderbilt University
Nashville, TN 37235
{a.gokhale@vanderbilt.edu}

Jeff Gray
Dept. of Computer
and Information Science
Univ. of Alabama at Birmingham
Birmingham, AL
{gray@cis.uab.edu}

Abstract

Some of the most challenging problems in high performance computing are those associated with producing software for distributed, high performance software (DHPS) systems in which distributed resources, such as processors, sensors and networks coordinate to address high performance computing needs of different domains. Supporting the multiple simultaneous quality of service (QoS) and functional requirements of DHPS systems is particularly vexing for DHPS system developers and integrators, who must address these QoS and functional requirements without overcomplicating their solutions, degrading software quality, and exceeding project time and effort constraints. A key enabler in recent successes with small- to medium-scale DHPS systems has been middleware, which provides reusable building blocks. However, as DHPS systems scale to form large "composable systems of systems," the design space comprising available choices of reusable building blocks and services in middleware grows substantially. This limits the ability of the DHPS developers to make the right design choices, which adversely impacts validation and verification of performance of end systems, and hence project costs.

To enable DHPS developers to make the right design choices, a systematic methodology to analyze the performance of DHPS 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. As a first step towards building this methodology, in this paper we present a model of the Reactor pattern, which provides the very important synchronous demultiplexing and dispatching capabilities in DHPS systems. The model is based on the Stochastic Reward Net (SRN) modeling paradigm. We illustrate how the model could be used relatively easily to obtain estimates of key performance metrics and for the sensitivity analysis of the Reactor pattern.

1. Introduction

Emerging Trends and Challenges

Large-scale, distributed, high performance software (DHPS) systems form the basis of numerous scientific grid computing applications. DHPS systems comprise many interdependent artifacts, such as network/bus interconnects, many coordinated local and remote endsystems, and multiple layers of software. DHPS systems demand multiple simultaneous quality of service (QoS) properties including predictability, controllability, and adaptability of operating characteristics for applications with respect to such features as time, throughput, accuracy, confidence, security and synchronization. All these issues become highly volatile in large-scale DHPS systems, due to the dynamic interplay of the many interconnected parts that are often constructed from smaller parts.

Although it is possible in theory to develop these types of complex DHPS systems from scratch, contemporary economic and organizational constraints, as well as increasingly complex requirements and competitive pressures, make it infeasible to do so in practice. A key enabler in recent successes with DHPS systems has been *QoS-enabled middleware* [20], such as Globus [3].

Middleware comprises software layers that provide platform-independent execution semantics and reusable services that coordinate how application components are composed and interoperate. The primary role of middleware is to (1) functionally bridge the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate, (2) enable and simplify the integration of components developed by multiple technology suppliers, and (3) provide common reusable accessibility for functionality and patterns by factoring out artifacts from applications into reusable code.

The flexibility and configurability offered by middleware is manifested in the large number of reusable software building blocks and configuration options, which can be used to compose and build large systems end to end. These building blocks embody good design practices called patterns [4, 22]. The choice of the pattern to use is dependent on the context and the consequences of using the pattern.

Current *ad hoc* techniques based on manually choosing the right set of building blocks are error-prone and may adversely impact performance, system costs and schedules, since most errors are caught very late in the lifecycle of DHPS development. It is desirable to have the ability to analyze the performance of individual building blocks and the composed system much earlier in the lifecycle of DHPS systems, thereby significantly lowering system testing costs as well as improving correctness of the final developed system.

To address the challenges of performance evaluation of DHPS systems design, a systematic performance analysis methodology is necessary. This methodology would comprise performance models of the individual building blocks and the composition of the building blocks. The performance models are based upon well-known analytical/numerical modeling paradigms [18, 2, 8] and simulation techniques [25]. As a first step towards the development of such a methodology, this paper presents a model of the Reactor pattern [22], which provides the very important synchronous demultiplexing and dispatching capabilities in DHPS systems. The model is based on the Stochastic Reward Net (SRN) modeling paradigm [18]. We illustrate how the model can be used to obtain estimates of key performance metrics and for the sensitivity analysis of the Reactor pattern with relative ease.

Paper Organization

This paper is organized as follows: Section 2 provides an overview of performance modeling techniques we use followed by related work; Section 3 presents the SRN model of the Reactor pattern, and illustrates the use of the model through examples; and finally Section 4 offers concluding remarks and directions for future research.

2. Background and Related Work

This section provides an overview of the Stochastic Reward Nets (SRN) analytical modeling techniques we have used for performance modeling of DHPS systems. We also describe the related work in this area.

2.1. Stochastic Reward Nets (SRNs)

This section describes the background on the Stochastic Reward Net (SRN) modeling paradigm [18] used for performance analysis of the Reactor pattern. SRNs represent a powerful modeling technique that is concise in its specification and whose form is closer to a designer's intuition about what a model should look like. Since SRN specification is closer to a designer's intuition of system behavior, 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 are an extension to Petri nets [15]. Petri nets were proposed to represent formally the flow of control in a system. A Petri net is a directed graph which comprises two types of elements, namely, *places* and *transitions*. A directed arc connecting a place to a transition is called an *input arc*. Conversely, an *output arc* connects a transition to a place. Arcs have a positive integer number called *multiplicity* associated with them, with the default multiplicity associated with an arc being 1. 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 arcs 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 one of the corresponding input places and a number of tokens equal to the output arc multiplicity is deposited in each one of the corresponding output places. The state of a Petri net with p places is represented by a vector $(m_1, m_2, \ldots, m_p)$, where m_i is the number of tokens in place i. The state of a Petri net is often referred to as its marking. When a Petri net is specified, it can be made to start at a particular marking called the *initial marking*. Subsequently, the net evolves by the successive enabling and firing of transitions which causes the tokens to flow among places.

Stochastic Petri nets [18] extend Petri nets by allowing timed transitions that have *exponentially distributed* firing times. Generalized stochastic Petri nets (GSPNs) [18] also allow *immediate* transitions which fire instantaneously. GSPNs 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 default multiplicity is 1. A GSPN 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*.

Stochastic reward nets (SRNs) extend GSPNs further by allowing the association of a reward rate to each tangible marking. The SRN models allow the concise specification of various reward functions. To extend the power of specification, SRN includes specification of *enabling (or guard) functions* for each transition. The transition is enabled only if the enabling function returns "1." SRNs also allow marking dependent arc multiplicities and enabling functions. Another feature of SRNs is the provision of priorities and probabilities to determine which of a set of simultaneously enabled transitions will fire first: the transition with the highest priority is fired first. If the competing transitions have the same priority the one to fire first is chosen probabilistically.

In a graphical representation of a SRN, a place is represented as a circle, n tokens in a place are represented by n dots or the number n within the place, immediate transitions are represented by thin lines, and exponentially distributed timed transitions are represented by empty rectangles. An inhibitor arc is represented by a circle instead of an arrow at the terminating end. An arc with multiplicity m is represented by a "$|m$" on the arc, and an arc with a marking dependent multiplicity function is indicated by a "N" or an inverted "N" in it. The number of tokens in place p is indicated as p.

2.2. Related Work

Stochastic reward nets have been extensively used for performance, reliability and performability analysis of a variety of systems [19, 9, 10, 24, 11, 14]. The work closest to the proposed research is reported by Ramani *et al.* [19], where SRNs are used for the performance analysis of the CORBA event service. UML representations of application architecture have been mapped to Queuing networks [26, 7, 13, 16, 17] and Petri nets [12, 1].

Several other analysis tools exist. The Virginia Embedded System Toolkit (VEST) [23] is a model-based embedded system composition tool that checks whether certain real-time, memory, power, and cost constraints of DPSS applications are satisfied. The Cadena [5] tool suite provide static analysis, model-checking, and other light-weight formal methods for component middleware-based DPSS systems. While these tools are important, they either deal with other systems features (*e.g.*, power consumption) or have a narrow focus (*e.g.*, embedded systems). Moreover, these tools are applicable once the system composition decisions are made.

3. Performance Evaluation of the Reactor Pattern

In this section we describe the process of constructing a SRN model for the Reactor pattern. Towards this end, we first provide an overview of the Reactor pattern and describe its characteristics and the relevant performance measures. A SRN model of the Reactor pattern is presented along with a discussion of how the performance measures can be obtained by assigning reward rates at the net level. We conclude the section with illustrative examples of how the SRN model can be used to obtain an estimate of the performance metrics as well as to analyze the sensitivity of the performance metrics for different values of the input parameters and configuration options.

3.1. 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 the correct application-supplied event handler. This is the idea behind the Reactor pattern, which provides synchronous event demultiplexing and dispatching capabilities.

The Reactor pattern could be implemented in many different ways depending on the event demultiplexing capabilities provided by the operating systems 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 giving rise to a single-threaded Reactor implementation. Alternately, the event could be delegated to a pool of threads to handle the events to give rise to a thread-pool Reactor.

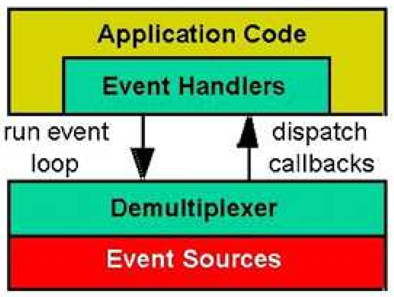

Figure 1. Event Demultiplexers in Middleware

3.2. 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.
- Events of type #1 are serviced with a higher priority over events of type #2.
- Event arrivals for both types of events follow a Poisson distribution 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, when event handles corresponding to both event types are enabled, the event corresponding to type #1 is serviced with a priority over event handle of type #2 event.

3.3. Performance Metrics

The following performance metrics are of interest for each one of the event types in the Reactor pattern described in Section 3.2:

- **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 since it is possible to develop appropriate scheduling policies for applications with real-time requirements.
- **Expected total number of events** – which provides an estimate of the total number of events in a system. These estimates are also tied to scheduling decisions. In addition, these estimates will determine the right levels of resource provisioning required to sustain the system demands.
- **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. Alternately, these also give an estimate on the desired levels of resource provisioning.

3.4. SRN Model

Figure 2 shows the SRN model for the Reactor pattern with the characteristics described in Section 3.2. Part (a) models the arrival, queuing and service of the two types of events, where 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 transition $Sr2$ 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 the events of type #1 over events of type #2.

Part (b) of the net 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

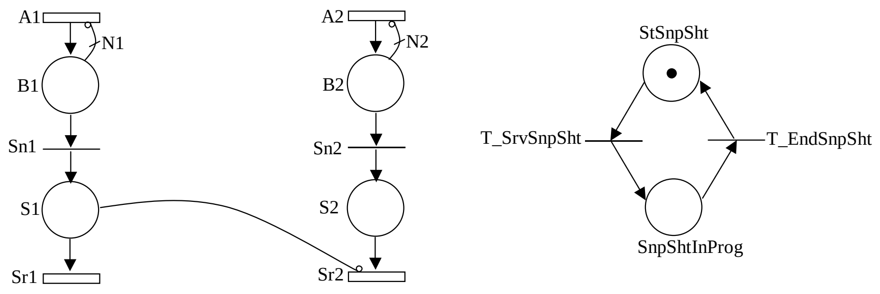

Figure 2. SRN model for the Reactor pattern

current snapshot are being serviced. Once 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 summarizes the enabling/guard functions for the transitions in the net.

The process of taking a single snapshot is modeled by the example SRN model presented in Figure 2. The example figure considers the 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 of 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 the 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 one 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 two 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 3.3 can be computed by assigning reward rates at the net level as summarized in Table 2. The throughputs T_1 and T_2 are given by the rate at which transitions $Sr1$ and $Sr2$ fire. The queue lengths Q_1 and Q_2 are given by the number of tokens in places $B1$ and $B2$, respectively. The total number of events E_1 is given by the sum of the number of tokens in places $B1$ and $S1$. Similarly, the total number of events E_2 is given by the sum of the number of tokens in places $B2$ and $S2$. The loss probability L_1 is given by the probability of $N1$ tokens in place $B1$. Similarly, the loss probability L_2 is given by the probability of $N2$ tokens in place $B2$.

3.5. Illustration

This section illustrates how the SRN model presented in Section 3.4 can be used to determine the impact of different parameters on the performance measures by careful design of experiments. The SRN is solved using SPNP [6] to obtain the expected values of the performance measures in all of the experiments described below.

In the first experiment, the impact of buffer capacity is determined on the performance measures. For this experi-

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

Table 1. Guard functions

Performance metric	Notation	Reward rate
Throughput of event type #1	T_1	return rate($Sr1$)
Throughput of event type #2	T_2	return rate($Sr2$)
Queue length of event type #1	Q_1	return ($\#B1$)
Queue length of event type #2	Q_2	return ($\#B2$)
Loss probability of event type #1	L_1	return ($\#B1 == N1?1:0$)
Loss probability of event type #2	L_2	return ($\#B2 == N2?1:0$)
Total number of events of type #1	E_1	return($\#B1 + \#S1$)
Total number of events of type #2	E_2	return($\#B2 + \#S2$)

Table 2. Reward assignments to obtain performance measures

ment, the values of the remaining parameters (except for the buffer capacities) are summarized in Table 3. We consider two values of buffer capacities N_1 and N_2. In the first experiment, there is a buffer capacity of 1 for both types of events, whereas the second experiment offers a buffer capacity of 5 for both types of events. The performance metrics for both of the experiments are summarized in Table 4. Because the values of the parameters of the first type of events (λ_1, μ_1 and N_1) are the same as the values of the parameters for the second type of events (λ_2, μ_1, and N_2), the throughputs, queue lengths, and the loss probabilities are the same for these two event types.

The total number of events for the events of type #2, denoted E_2 is slightly higher than the total number of events of type #1, denoted E_1. Because the events of type #1 are provided prioritized service over the events of type #2, on average it takes longer to service a type #2 event than it takes to service a type #1 event. This results in a higher total number of events of the second type than of the first type. These observations hold for both values of maximum buffer capacity. It can be observed that the loss probability is 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 maximum buffer capacity is 1 than when the maximum buffer capacity is 5.

The sensitivity of the performance measures to the arrival rate of the events of type #1 and type #2 can be determined; i.e., λ_1 and λ_2. For sensitivity analysis, the maximum buffer capacity can be set for both event types to be 5.

Parameter	Value
λ_1	0.400/sec.
λ_2	0.400/sec.
μ_1	2.000/sec.
μ_2	2.000/sec.

Table 3. Values of parameters

Performance measure	Buffer space	
	$N_1 = 1$, $N_2 = 1$	$N_1 = 5$, $N_2 = 5$
T_1	0.37/sec.	0.40/sec
T_2	0.37/sec.	0.40/sec
Q_1	0.065	0.12
Q_2	0.065	0.12
E_1	0.25	0.32
E_2	0.27	0.35
L_1	0.065	0.00026
L_2	0.065	0.00026

Table 4. Impact of buffer capacity on performance measures

Variations can be applied to both λ_1 and λ_2 one at a time in the range of 0.5/sec. to 2.0/sec. to obtain the expected values of the performance metrics by solving the SRN model shown in Figure 2. The remaining parameters are held at the values reported in Table 3.

Figure 3 shows the performance measures as a function of λ_1 and λ_2. The plots in the left column show the variation of the performance measures with respect to λ_1, whereas the plots in the right column show the variation of the performance measures with respect to λ_2. Referring to the topmost figure in the left column, it can be observed that initially, the throughput of type one events is nearly the same as the arrival rate of the events, indicating that the events are serviced at the same rate at which they arrive. However, as λ_1 increases, the throughput starts lagging the arrival rate, which indicates that the service rate μ_1 is not sufficiently high to process the events at the rate at which they arrive. This may cause the event queue for type #1 events to operate at full capacity for an extended period of time, which results in a rejection of the incoming input events.

Thus, a decrease in the rate at which the throughput increases is marked by an increase in the loss probability of the events (as shown in the third plot in the left column) and an increase in the queue length (as shown in the fourth plot in the left column) in Figure 3. The left plot in the bottom row represents the total number of events of each type as a function of λ_1. The plot indicates that the total number of type #2 events in the system increases as λ_1 increases. When λ_1 increases, the probability of having to service a type #1 event prior to servicing a type #2 event in a given snapshot increases. Because events of type #1 have priority over events of type #2, the type #2 events tend to reside longer in the system as λ_1 increases. Thus, although the throughput of type #2 events is unchanged with respect to λ_1, the response time of a type #2 event may increase. The plots in the right column of Figure 3 indicate that similar trends can be observed in the performance measures with respect to λ_2, except that the roles of the two types of events are reversed.

Similar sensitivity analysis can also be conducted with respect to the service rates μ_1 and μ_2.

4. Conclusion and Future Research

The research in this paper is motivated by the realization that a growing number of computing and networking resources are being expended to control large-scale, distributed, high performance software (DHPS) systems. The next-generation of DHPS systems, such as sensor network applications, grid-based scientific experiments, and emergency response systems, evolve rapidly and must collaborate with multiple remote sensors, provide on-demand browsing and actuation capabilities for human operators, and respond flexibly to unanticipated situational factors that arise at run-time. These characteristics render earlier static system development and analysis techniques less effective. Moreover, the availability of off-the-shelf software, hardware, and networking building blocks – compounded by economic and market forces – are causing DHPS systems to be assembled rapidly and tested by composing building block components. Design-time validation and verification of end system performance is a necessity for the realm of next-generation DHPS systems.

In this paper, we provided an overview of the SRN modeling paradigm that may be valuable in developing a methodology for design time performance analysis. Further, we presented a model of an important software building block called the Reactor, which is used in many DHPS systems. We then demonstrated how the SRN model can be used to obtain estimates of key performance metrics of the Reactor, as well as to analyze the sensitivity of the performance metrics to the values of the input parameters and different configuration options with relative ease. Such results, obtained earlier in the system lifecyle (*e.g.*, design-time), can enable DHPS developers in making informed decisions regarding appropriate resource provisioning and schedulability.

In the near term, our future work will involve validating the performance metrics of the Reactor obtained from the SRN model using empirical benchmarking. Empirical benchmarking is relatively complex owing to the difficulties of testing the Reactor as a standalone unit without the confounding effects of underlying OS demultiplexing and queueing mechanisms. For example, in POSIX-compliant operating systems, there will always be a default buffering capability associated with the socket handles on which the `select()` system call is invoked. As a result, it is cumbersome to showcase a Reactor with a configurable queue size for each of its input event handles. Validating the Reactor model may require development and composition of the models for other building blocks with which the Reactor interacts. Developing performance models for other versions of the Reactor, including the *thread pool* [21] Reactor, is also a topic of future research.

In the long term, we expect to develop and validate a suite of performance models for most of the middleware building blocks, as well as validation of the compositions of these building blocks.

References

[1] A. Bondavalli, I. Mura, and I. Majzik. Automated dependability analysis of UML designs. In *Proc. of Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1998.

[2] H. Choi, V. Kulkarni, and K. S. Trivedi. “Markov Regenerative Stochastic Petri Net”. *Performance Evaluation*, 20(1–3):337–357, 1994.

[3] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

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

[5] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

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

[7] F. Hoeben. "Using UML models for performance calculation". In *Proc. of Workshop on Software and Performance*, 2000.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] P. King and R. Pooley. "Derivation of Petri net performance models from UML specifications of communication software". In *Proc. of XV Performance Engineering Workshop*, 1997.

[13] D. A. Menasce and H. A. Gomaa. "On a language based method for software performance engineering of client/server systems". In *Proc. of Workshop on Software and Performance*, 1998.

[14] 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.

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

[16] D. Petriu, C. Shousha, and A. Jalnapurkar. "Architecture–based performance analysis applied to a telecommunication system". *IEEE Trans. on Software Engineering*, November 2000.

[17] R. Pooley and P. King. "The unified modeling language and performance engineering". *IEEE Software*, 1999.

[18] 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.

[19] 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.

[20] R. E. Schantz and D. C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In J. Marciniak and G. Telecki, editors, *Encyclopedia of Software Engineering*. Wiley & Sons, New York, 2002.

[21] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts, 2002.

[22] 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.

[23] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An Aspect-based Composition Tool for Real-time Systems. In *Proceedings of the IEEE Real-time Applications Symposium*, Washington, DC, May 2003. IEEE.

[24] 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.

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

[26] L. G. Williams and C. U. Smith. "Performance evaluation of software architectures". In *Proc. of the Workshop on Software and Performance*, Santa Fe, NM, 1998.

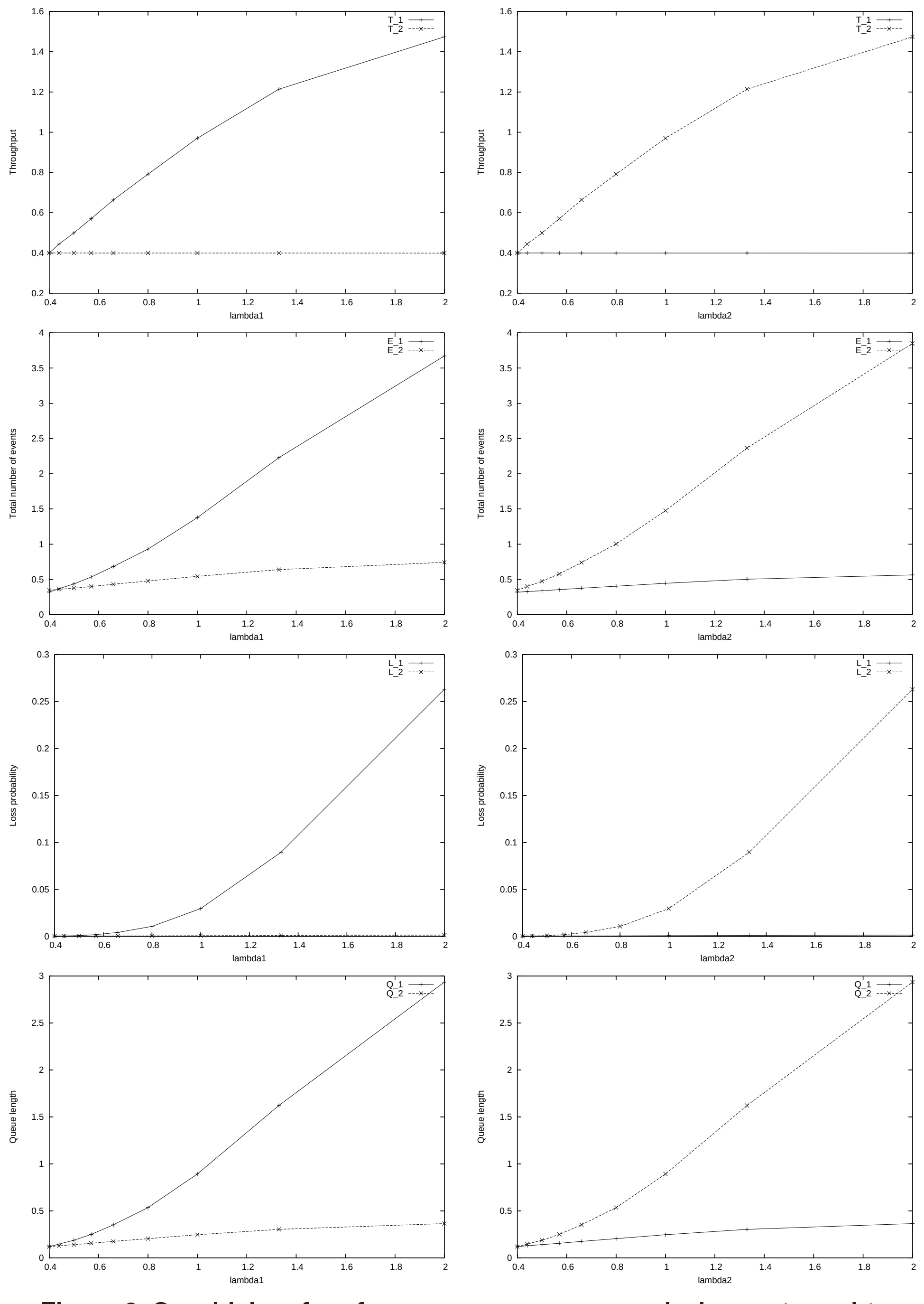

Figure 3. Sensitivity of performance measures to arrival rates λ_1 and λ_2