Performance Evaluation of Middleware Event Demultiplexing Patterns in Distributed Performance-Sensitive Software Systems

Authors
Addresses
Emails

ABSTRACT

Some of the most challenging problems facing the information technology profession are those associated with producing software for distributed, performance-sensitive software (DPSS) systems in which distributed resources, such as processors, sensors and networks coordinate to control physical, chemical, or biological processes or devices. Supporting multiple simultaneous quality of service (QoS) and functional requirements of DPSS systems is particularly vexing for DPSS 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 DPSS systems (such as avionics systems) has been middleware. However, as DPSS systems scale to form large "composable systems of systems", the design space comprising available choices of software patterns embodied in reusable middleware frameworks and services grows substantially. This limits the ability of the DPSS developers to make the right design choices, which adversely impacts validation and verification of performance and correctness of end systems, and hence project costs.

This paper provides three contributions to the performance evaluation of middleware-based composable DPSS systems. First, we outline the challenges faced by DPSS developers in making the right middleware design choices. Second, we elicit the need to conduct performance analysis of reusable building blocks and subsequently the application as a whole prior to the actual composition of the application. Third, we describe a model of the Reactor pattern, which provides the very important synchronous demultiplexing and dispatching capabilities in DPSS systems. The model is based on the stochastic reward net modeling paradigm. We illustrate how the model could be used to obtain estimates of key performance metrics and sensitivity analysis of the Reactor pattern.

1. INTRODUCTION

Emerging Trends and Challenges

Large-scale, distributed, performance-sensitive software (DPSS) systems form the basis of mission- and often safety-critical applications, such air traffic control, industrial process automation, nuclear reactors, oil refineries, power grids, telecommunication networks, inventory management systems and patient monitoring systems. Figure 1 illustrates a sampling of such DPSS systems.

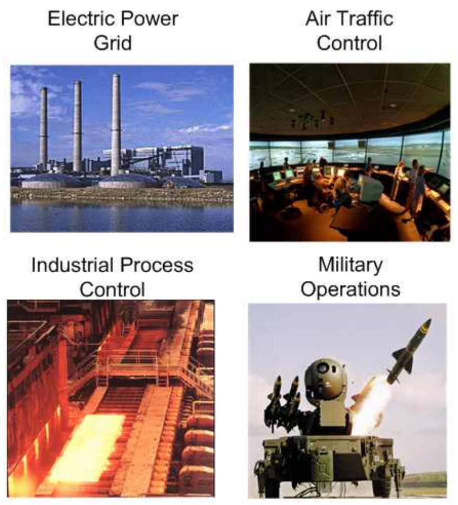

Figure 1: A Sampling of Large-scale Distributed Performance-sensitive Software-based Systems

DPSS systems comprise many interdependent artifacts, such as network/bus interconnects, many coordinated local and remote endsystems, and multiple layers of software. DPSS 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, quantity of information, accuracy, confidence, and synchronization. All these issues become highly volatile in large-scale DPSS 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 DPSS 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 small- to medium-scale DPSS systems (such as avionics mission computing or enterprise warehouse management systems) has been *QoS-enabled middleware* [15], as shown in Figure 2.

Middleware comprises software layers that provide platform-independent

Submitted to Sigmetrics 2005 Jun. 6-10, 2005, Banff, Alberta, Canada.

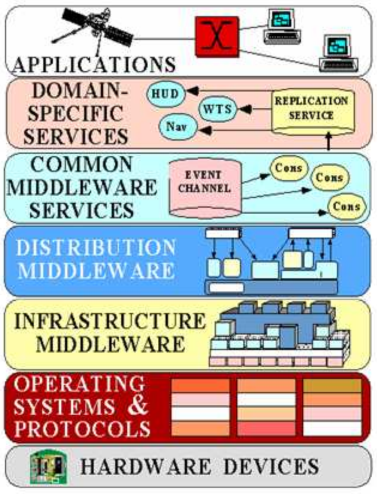

Figure 2: Middleware Stack

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 a 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 [2, 17] and may often provide similar functionality but are only suitable in specific operational contexts. A usage of these frameworks in arbitrary contexts may lead to significant performance overheads and may even impact the end system's correctness. The magnitude and complexity of DPSS systems, thus, demands stringent quality control and testing.

Today's *ad hoc* techniques based on manually making the choices of the building blocks to build DPSS systems are error-prone and adversely impact performance, system costs and schedules, since most errors are caught very late in the lifecycle of DPSS systems. 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 DPSS systems thereby significantly lowering system testing costs as well as improving correctness of the final developed systems.

To address the challenges of performance evaluation of DPSS systems design, this paper describes our research in applying the well-established modeling and analysis paradigm of stochastic reward nets (SRNs) [12] to middleware building blocks. In particular, we showcase our performance models on the Reactor pattern [17], which is a synchronous event demultiplexing and dispatching framework available in distributed middleware technologies.

Paper Organization

This paper is organized as follows: Section 2 elaborates on the technical challenges in composing and developing the next generation of DPSS systems using patterns-based building blocks, and provides background on the Reactor pattern; Section 3 provides technical background on the analytical techniques used to evaluate performance of patterns-based design and provides related work; Section 4 describes the process of constructing a Markov model for the Reactor pattern. It then describes how the different performance metrics could be obtained by assigning appropriate reward rates to the states of the model. The process of constructing the Markov model is used to motivate the necessity of a high-level specification mechanism of SRNs for the automatic generation of the Markov model. A SRN model of the Reactor pattern is then presented, and through examples the use of the SRN model to obtain the different performance measures is illustrated; and finally Section 5 discusses the insights gained through this exercise evaluating the advantages and limitations of this approach.

2. TECHNICAL CHALLENGES DEVELOPING DPSS SYSTEMS

Section 1 discussed how the next generation of DPSS systems are increasingly being developed and deployed using reusable middleware building blocks. This section elaborates on the patterns-based building blocks provided by contemporary middleware technologies eliciting the technical difficulties DPSS developers will face when constructing these systems. Furthermore, we provide details on the Reactor [17] pattern, which is the focus for performance evaluation in this paper.

2.1 Patterns-based Building Blocks in Middleware

Figure 3 illustrates a family of patterns forming a pattern language [1] for DPSS systems. A software pattern codifies recurring solutions to a particular problem occurring in different contexts, which is embodied as a reusable software building block. When designing DPSS systems, its developers will be required to decide the right set of patterns to use in the system composition, such that the desired end to end QoS and functional requirements are met. The choice of the patterns to use are driven by various factors including the context in which the DPSS system will be deployed, concurrency and distribution requirements of the application, and other QoS concerns, such as end-to-end latency and timeliness requirements for real-time DPSS systems or throughput for other applications (e.g., a telecommunication call processing switch).

Figure 3 depicts a number of interacting patterns. Patterns, such as the *Active Object* and *Leader-follower* [17] provide alternate approaches to concurrency with each solution having its own advantages and disadvantages. For example, although the Active Object pattern is simple to implement, it incurs an additional performance penalty due to thread context switching and message queueing. On the other hand, the Leader-Follower does not incur these drawbacks, but its implementation is very complex and might lead to race conditions. Other patterns like the *Reactor* and *Proactor* [17] provide solutions to synchronous and asynchronous event demultiplexing and dispatching, respectively. Both of these patterns provide the building block of the asynchronous part of the *half-sync/-half-async* pattern shown in Figure 3. Patterns, such as the *Asynchronous Completion Token* pattern can be used only in those cases where the underlying platform can support asynchronous messaging. Developers of DPSS systems need to make the right choices

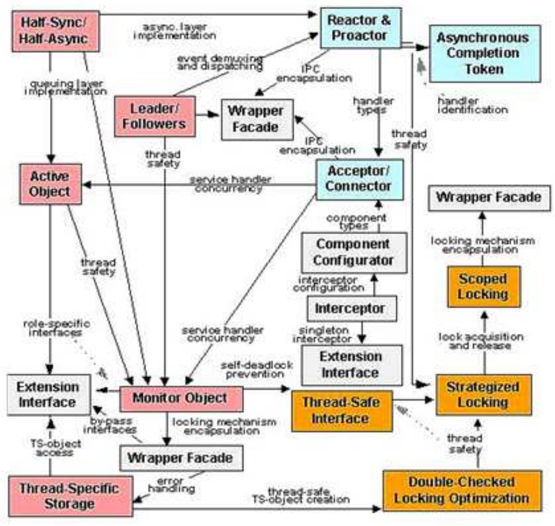

Figure 3: Middleware Patterns and Pattern Languages

of patterns to use in their solutions, such that the end-to-end QoS and functional requirements of the DPSS systems are met.

Contemporary middleware technologies provide these reusable building blocks to the DPSS developers. However, the responsibility of making the right choice rests on the developers. Moreover, without any estimates of expected end-to-end performance provided by the middleware, DPSS developers are forced to rely on rigorous testing later in the lifecycle to determine if the design choices were appropriate. Unfortunately, these decision occur very late in the lifecycle of DPSS systems, which adversely impacts costs and schedules. A mechanism is needed that will enable performance estimation of composable systems at design time. It is well understood that any changes at design-time are much easier to incorporate into the system. Moreover, the higher-levels of abstraction and formalism provided by the design-phase makes it possible to formally verify the systems for correctness.

Model-driven Software Development (MDSD) [10, 8] has gained prominence in assisting DPSS developers to make the right choices in designing and composing large systems. Model-based performance analysis can be used to analyze the performance of patterns-based DPSS systems. In order to facilitate the use of model-based performance analysis, however, it is necessary for MDSD tools to provide analytical models of the various possible patterns. These models can be used to estimate the performance of each pattern when used in different configurations and contexts. Subsequently, these models can be used in a plug-and-play fashion to form larger models corresponding to composition of patterns in the DPSS system. This paper focuses on using Stochastic Reward Nets (SRNs) [12] as a paradigm to model middleware patterns. Our technique is showcased in Section 4.4 in the context of the Reactor pattern [17], which is described briefly in Section 2.2.

2.2 Reactor Pattern and Middleware Implementations

Figure 4 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 occurrence of an event, the demultiplexer dispatches the event by making a callback to the right application-supplied event handler. This is the idea behind the Reactor pattern, which provides synchronous event demultiplexing and dispatching capabilities.

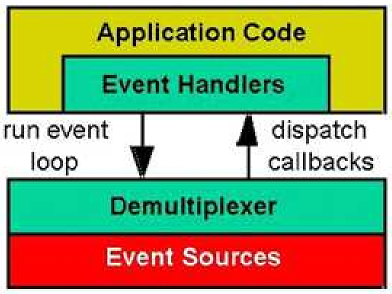

Figure 4: Event Demultiplexers in Middleware

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 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. Delegation of events to one of the threads of the thread pool can be implemented using either the *Leader-Follower* or the *Active Object* patterns mentioned earlier. These different choices are dictated by the requirements of the application.

3. STOCHASTIC REWARD NETS (SRNS)

This section describes the technical background on the modeling paradigms used for performance evaluation in this paper, with a focus on Stochastic Reward Nets (SRN) [12]. 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. This circumvents the process of manual construction of a Markov reward model, which is cumbersome and error-prone even for a system with moderate number of states. 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. The rest of the section describes SRNs and related modeling techniques.

SRNs are an extension to Petri nets [11]. Petri nets were proposed to formally represent 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 [12] extend Petri nets by allowing timed transitions that have *exponentially distributed* firing times. Generalized stochastic Petri nets (GSPNs) [12] 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 extend GSPNs further by allowing the association of a reward rate to each tangible marking. The tangible markings of an SPN, GSPN, or an SRN and their rates of transition from one marking to another are in fact equivalent to corresponding states and transitions between states of an underlying continuous time Markov chain (CTMC). Hence, an SRN can be mapped into an equivalent Markov Reward Model (MRM) [3]. Packages such as SPNP [4] can automate the translation of an SRN into its equivalent MRM and solve it. 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". SRN also allows marking dependent arc multiplicities and enabling functions. Another feature of SRN 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.

Stochastic reward nets have been extensively used for the performance, reliability and performability analysis of a variety of systems [13, 5, 6, 18, 7, 9]. The work closest to the research described in this paper is reported by Ramani *et al.* [13], where SRNs are used for the performance analysis of the CORBA event service to obtain the different performance metrics, such as the expected queue length and the probability of message loss.

4. PERFORMANCE EVALUATION OF THE REACTOR PATTERN

This section describes the process of constructing different performance evaluation models for the Reactor pattern and provides analytical performance results solving these models. Initially, a Markov model for the Reactor pattern is presented. Towards this end, we first describe the characteristics of the analytical model for the Reactor pattern and the relevant performance measures. We then describe how these performance measures can be obtained from the Markov model by an appropriate assignment of the reward rates. The complex (and potentially error-prone) process of constructing a Markov model serves as a motivation for a high-level specification mechanism of Stochastic Reward Nets (SRNs) for the automatic generation of the Markov model. 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 configuration options.

4.1 Characteristics of the Reactor Pattern

This paper focuses on the analytical modeling of the Reactor pattern. To demonstrate our techniques, we have chosen the simpler single-threaded, *select*-based Reactor implementation. In our model we consider a Reactor pattern with the following characteristics:

- The system is modeled with 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 one events is denoted N_1 and of type two 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 one is serviced with a priority over event handle of type two event. over event

4.2 Performance Metrics

The following performance metrics are of interest for each one of the event types in the Reactor pattern described in Section 4.1:

- **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 applications. such as telecommunications call processing.

- **Expected queue length** – which provides an estimate of the queueing 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 making 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.
- **Response time:** – which indicates the time taken to service an incoming event. This estimate is important to

Section 4.3 describes the performance analysis of the Reactor pattern using a Markov Reward Model. Section 4.4 describes the performance analysis of the Reactor pattern using a Stochastic Reward Net.

4.3 Performance Evaluation using Markov Reward Model

In this section we describe the construction of the Markov model for the Reactor pattern described in Section 2.2 for the characteristics and policies described in Section 4.1. To illustrate the reasoning used in the construction of the model, we consider a simple case where the maximum buffer capacity for each type of event is one, where $N_1 = 1$, and $N_2 = 1$.

The state of the system in this case is described by a 4-tuple (i, j, k, l). The first element, i, in the tuple represents the number of events in the queue of the first event type; the second element, j, represents the number of events in the queue of the second event type; the third element in the tuple, k, represents an enabled handle serving the first type of event; and the fourth element, l, represents an enabled handle corresponding to the second type of event. Figure 5 shows the Markov reward model for the Reactor pattern when the maximum buffer capacity for each event type is assumed to be one.

The evolution of the system through the sequence of states is described below.

- **Initial condition:** Initially, the system starts in state $(0, 0, 0, 0)$, which depicts the condition that no events are queued and no event handles are enabled. In the initial state $(0, 0, 0, 0)$, any one of the two events can occur, namely, the arrival of the first type of event or the arrival of the second type of event.
- **Event arrivals:** Upon the occurrence of either one of these events described above, a snapshot is taken, and the incoming event is serviced. Thus, the occurrence of the first type of event transitions the system to state $(0, 0, 1, 0)$ from $(0, 0, 0, 0)$, whereas, the occurrence of the second type of event transitions the system to state $(0, 0, 0, 1)$ from state $(0, 0, 0, 0)$. Taking a snapshot is equivalent to the Reactor listening for an event to be notified of the event and the ensuing operations of demultiplexing and dispatching described in Section 2.2. In state $(0, 0, 1, 0)$ if an event of type one arrives then it is

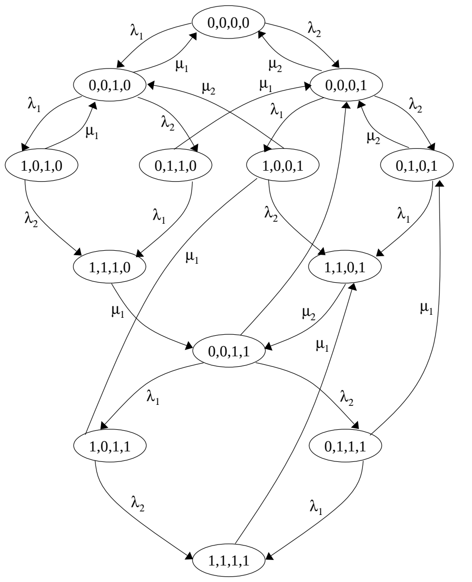

Figure 5: Markov model for the Reactor pattern

queued and the system transitions to state $(1, 0, 1, 0)$. Similarly, the occurrence of an event of type two in state $(0, 0, 1, 0)$ transitions the system to state $(0, 1, 1, 0)$. Using similar reasoning, the system can transition from state $(0, 0, 0, 1)$ to state $(1, 0, 0, 1)$ and $(0, 1, 0, 1)$. In state $(1, 0, 1, 0)$ an incoming event of type two is queued which transitions the system to state $(1, 1, 1, 0)$. Similarly, in state $(0, 1, 1, 0)$ an incoming event of type one is queued which transitions the system to state $(1, 1, 1, 0)$. In state $(0, 0, 1, 1)$, if a first type of event arrives, the system transitions to state $(1, 0, 1, 1)$, if a second type of event arrives, the system transitions to state $(0, 1, 1, 1)$. In state $(1, 0, 1, 1)$, an arrival of second type of event transitions the system to state $(1, 1, 1, 1)$.
- **Loss of events:** In state $(1, 0, 1, 0)$ an incoming event of type one is lost because the buffer is full. Similarly, in state $(0, 1, 1, 0)$, an incoming event of type two is lost. In states $(1, 1, 1, 0)$, $(1, 1, 0, 1)$ and $(1, 1, 1, 1)$ incoming events of both types are lost.
- **Handling priorities:** In states $(0, 0, 1, 1)$, $(1, 0, 1, 1)$, $(0, 1, 1, 1)$, and $(1, 0, 1, 1)$ the handles corresponding to both types of events are enabled. However, since the event of type one is serviced with a higher priority than the event of type two, the only service completion that can occur in these states is that of type one event. Upon the completion of service of first type of event, the system transitions to state $(0, 0, 0, 1)$ from state $(0, 0, 1, 1)$, to state $(1, 0, 0, 1)$ from state $(1, 0, 1, 1)$, to state $(0, 1, 0, 1)$ from state $(0, 1, 1, 1)$ and to state $(1, 1, 0, 1)$ from state $(1, 1, 1, 1)$.
- **Service completions:** The only event that can occur in state $(1, 1, 1, 0)$ is the completion of service of first type of event. This marks the completion of the present snapshot and commencement of the next one, transitioning the system to state $(0, 0, 1, 1)$. Similarly in state $(1, 1, 0, 1)$, the completion of service of the second type of event transitions the system to state $(0, 0, 1, 1)$. Using similar arguments it can be seen that service completion from: (i) states $(0, 0, 0, 1)$ and $(0, 0, 1, 0)$ transition the system to the initial state $(0, 0, 0, 0)$. (ii) states

$(1,0,1,0)$ and $(1,0,0,1)$ transition the system to state $(0,0,1,0)$. (iii) states $(0,1,1,0)$ and $(0,1,0,1)$ transition the system to state $(0,0,0,1)$. (iv) state $(1,0,1,0)$ transition the system to state $(0,0,1,0)$ and (v) state $(0,1,0,1)$ transition the system to state $(0,0,0,1)$.

The Markov model presented in Figure 5 can be solved to obtain the probabilities of being in each one of the states using packages, such as SHARPE [14]. We let $p_{i,j,k,l}$ denote the probability of being in state (i,j,k,l). Next, we describe how the Markov reward model can be used to obtain the performance metrics described in Section 4.2 by appropriately assigning reward rates to the states in the model.

Throughput: Let T_1 and T_2 denote the expected throughput for the first and second type of events respectively. T_1 is the rate at which the events of type one are serviced, and is given by μ_1 times the sum of the probabilities of the states in which an event of type one is being serviced, namely, $k = 1$. Similarly, T_2 is the rate at which the events of type two are serviced, and is given by μ_2 times the sum of the probabilities of the states in which an event of type two is serviced, namely, $l = 1$ and $k = 0$.

The condition $k = 0$ is necessary to enforce the prioritized service provided to type one events over type two events. T_1 and T_2 are given by Equation (1) and (2). Thus, T_1 can be obtained by assigning a reward rate of μ_1 to states (i,j,k,l), where $k = 1$. Similarly, T_2 can be obtained by assigning the reward rate of μ_2 to those states where $l = 1$ and $k = 0$.

$$\begin{aligned} T_1 &= \mu_1 * (p_{0,0,1,0} + p_{1,0,1,0} + p_{0,1,1,0} + p_{1,1,1,0} + \\ &\quad p_{0,0,1,1} + p_{1,0,1,1} + p_{0,1,1,1} + p_{1,1,1,1}) \end{aligned} \tag{1}$$

$$T_2 = \mu_2 * (p_{0,0,0,1} + p_{1,0,0,1} + p_{0,1,0,1} + p_{1,1,0,1}) \tag{2}$$

Queue length: Let Q_1 and Q_2 denote the queue lengths for the events of type one and two respectively. Typically, Q_1 is obtained as the weighted sum of the number of events of type one in the buffer, and the weights are given by the probability distribution of the number of events. Similarly, Q_2 is obtained as the weighted sum of the number of events of type two in the buffer. Since the maximum buffer capacity is one, Q_1 will be given by the sum of the probabilities of the states in which there is one event in the buffer of type one events, or $i = 1$.

Similarly, Q_2 will be given by the sum of the probabilities of those states in which there is one event in the queue of type two events, or $k = 1$. Q_1 and Q_2 are given by Equations (3) and (4), respectively. Thus, Q_1 can be obtained by assigning a reward rate of 1.0 to the states in Equation (3). Similarly, Q_2 can be obtained by assigning a reward rate of 1.0 to the states in Equation (4).

$$Q_1 = 1.0 * (p_{1,0,1,0} + p_{1,1,1,0} + p_{1,0,1,1} + p_{1,1,1,1}) \tag{3}$$

$$Q_2 = 1.0 * (p_{0,1,0,1} + p_{1,1,0,1}) \tag{4}$$

Event loss probability: Let L_1 and L_2 be the loss probability of the events of type one and two, respectively. Typically, L_1 is the sum of the probabilities of those states in which the number of type one events in the queue is equal to the maximum buffer capacity N_1. Similarly, L_2 is the sum of the probabilities of those states in which the number of type two events in the queue is equal to the maximum buffer capacity N_2. Since the maximum buffer capacity for type one events is one, L_1 is the sum of the probabilities of the states in which $i = 1$, and L_2 is the sum of the probabilities of the states in which $k = 1$. L_1 and L_2 are also given by Equation (3) and Equation (4), respectively.

Total number of events: For each event type, the total number of events in the system is the sum of the enabled event handle and the number of events in the queue. The expected total number of events of type one and two (denoted E_1 and E_2) are given by Equation (5) and Equation (6), respectively. Thus, E_1 can be obtained by assigning a reward rate of 1.0 to states (i,j,k,l) where $i + k = 1$ and a reward rate of 2.0 to those states where $i + k = 2$. The condition $i + k = 1$ indicates that there is either one event of type one in the queue or one event handle enabled. The condition $i+k = 2$ indicates that there is an event of type one in the queue and an event handle corresponding to type one event enabled. Similarly, E_2 can be obtained by assigning a reward rate of 1.0 to states *i.e.*, $j + l = 1$, and a reward rate of 2.0 to states, *i.e.*, $j + l = 2$.

$$\begin{aligned} E_1 &= 1.0 * (p_{0,0,1,0} + p_{0,1,1,0} + p_{0,0,1,1} + p_{0,1,1,1}) + \\ &\quad 2.0 * (p_{1,0,1,0} + p_{1,1,1,0} + p_{1,0,1,1} + p_{1,1,1,1}) \end{aligned} \tag{5}$$

$$E_2 = 1.0 * (p_{0,0,0,1} + p_{1,0,0,1}) + 2.0 * (p_{0,1,0,1} + p_{0,1,1,1} + p_{1,1,1,1}) \tag{6}$$

Response time: We obtain the response time of the events using the tagged customer approach []. 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 response time of an event of an incoming type one and type two event is given by $1/\mu_1$ and $1/\mu_2$ respectively, 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.

An arriving event of type one can find the system in each one of the thirteen states shown in Figure 5. In the state $(0,0,0,0)$ when no event handle is being serviced, the waiting time is zero. In states $(1,0,1,0)$, $(1,0,0,1)$, $(1,1,1,0)$, $(1,1,0,1)$, and $(1,1,1,1)$ an arriving event of type one is lost. The waiting time of an arriving event of type one needs to be determined for all the remaining states. In state $(0,0,1,0)$, an event of type one is being serviced in the present snapshot when the tagged event of type one arrives.

Table 1: Conditional waiting time for type one events

State	Waiting time
$(0,0,0,0)$	0.0
$(0,0,1,0)$	$\frac{1}{\mu_1}$
$(0,0,0,1)$	$\frac{1}{\mu_2}$
$(0,1,1,0)$	$\frac{1}{\mu_1}$
$(0,1,0,1)$	$\frac{1}{\mu_2}$
$(0,0,1,1)$	$\frac{1}{\mu_1} + \frac{1}{\mu_2}$
$(0,1,1,1)$	$\frac{1}{\mu_1} + \frac{1}{\mu_2}$

Table 2: Conditional waiting time for type two events

State	Waiting time
$(0,0,0,0)$	0.0
$(0,0,1,0)$	$\frac{1}{\mu_1}$
$(0,0,0,1)$	$\frac{1}{\mu_1} + \frac{1}{\mu_2}$
$(1,0,1,0)$	$\frac{2}{\mu_1}$
$(1,0,0,1)$	$\frac{1}{\mu_1} + \frac{1}{\mu_2}$
$(0,0,1,1)$	$\frac{2}{\mu_1} + \frac{1}{\mu_2}$
$(1,0,1,1)$	$\frac{2}{\mu_1} + \frac{1}{\mu_2}$

This ongoing service must be completed, prior to the commencement of the service of the tagged event, and thus the waiting time in this state is $1/\mu_1$. In state $(0,0,0,1)$, an event of type two is being serviced in the current snapshot, and using the same argument as in the case of state $(0,0,0,1)$ the waiting time is $1/\mu_2$. In state $(0,1,1,0)$, an event of type one is being serviced. Although an event of type two is in the queue, in the next snapshot the incoming event of type one will be serviced prior to the event of type two that is queued due to the prioritized service provided to event of type one. Thus the waiting time in state $(0,1,1,0)$ is $1/\mu_1$. Using similar arguments, the waiting time in state $(0,1,0,1)$ is $1/\mu_2$. In state $(0,0,1,1)$, an event of type one is being serviced in the present snapshot, and the event handle corresponding to the event of type two is enabled. These two enabled event handles must be serviced prior to the commencement of the service of the tagged event, which results in a waiting time of $(1/\mu_1 + 1/\mu_2)$. In state $(0,1,1,1)$, due to the priority given to events of type one over event of type two, the waiting time is given by $(1/\mu_1 + 1/\mu_2)$. The conditional waiting times for a tagged event of type one are summarized in Table 1. The unconditional or the expected waiting time for an arriving event of type one is given by Equation (7). Using W_1, the expected response time for an event of type one is given by Equation (8).

$$\begin{aligned} W_1 \;=\; & \frac{1}{\mu_1} p_{0,0,1,0} + \frac{1}{\mu_2} p_{0,0,0,1} + \frac{1}{\mu_1} p_{0,1,1,0} + \\ & \frac{1}{\mu_2} p_{0,1,0,1} + (\frac{1}{\mu_1} + \frac{1}{\mu_2}) p_{0,0,1,1} + \\ & (\frac{1}{\mu_1} + \frac{1}{\mu_2}) p_{0,1,1,1} \end{aligned} \tag{7}$$

$$R_1 = \frac{1}{\mu_1} + W_1 \tag{8}$$

The waiting time for a tagged event of type two can be determined using the following reasoning. In the state $(0,0,0,0)$ when no event handle is being serviced, the waiting time is zero. On the other hand, in states $(0,1,0,1)$, $(0,1,1,0)$, $(1,1,1,0)$, $(1,1,0,1)$, $(0,1,1,1)$ and $(1,1,1,1)$ an incoming event of type two is lost. In state $(0,0,0,1)$, an event of type two is being serviced in the current snapshot, and this service must be completed. Prior to the service completion of this event which will mark the completion of the present snapshot, and the commencement of the service of the tagged event, two possible scenarios can occur. In the first scenario, no additional event of type one arrives in which case the waiting time is $1/\mu_2$. In the second scenario, an event of type one arrives and this type one event has to be serviced prior to the tagged event. Thus, the waiting time is $1/\mu_1 + 1/\mu_2$. The probabilities of these two scenarios can be computed and based on these probabilities the average waiting time can be determined. It can be seen, however, that the waiting time in case of the second scenario is pessimistic compared to the waiting time in case of the first scenario. Since an estimate of the response time will be used to determine if the performance of the system is capable of meeting real-time deadlines, we choose to consider the pessimistic waiting time in this scenario and in the rest of the paper. In state $(1,0,1,0)$, two events of type one (one that is already being serviced and the one that is in the queue) need to be serviced prior to the tagged event, and thus the waiting time is $2/\mu_1$. Using similar reasoning for the remaining states, the waiting time for a tagged event of type two can be determined for each one of the states and is summarized in Table 2. The expected waiting time for an incoming event of type two is given by Equation (9). The expected response time for an event of type two is given by Equation (10).

$$\begin{aligned} W_2 \;=\; & (\frac{1}{\mu_2} + \frac{1}{\mu_1})(p_{0,0,0,1} + \frac{2}{\mu_1} p_{1,0,1,0} + \\ & (\frac{1}{\mu_1} + \frac{1}{\mu_2}) p_{1,0,0,1} + (\frac{2}{\mu_1} + \frac{1}{\mu_2}) p_{0,0,1,1} + \\ & (\frac{2}{\mu_1} + \frac{1}{\mu_2}) p_{1,0,1,1} \end{aligned} \tag{9}$$

$$R_2 = \frac{1}{\mu_2} + W_2 \tag{10}$$

4.4 Performance Evaluation using Stochastic Reward Net

The manual construction of the Markov reward model becomes cumbersome and prone to errors as the maximum buffer capacities for the events and number of handlers increase. We now describe how SRNs can be used to specify the behavior of the Reactor pattern at a higher level, which is closer to the designer's intuition.

Figure 6 shows the SRN model for the Reactor pattern when the events of type one are serviced with priority over events of type two. It consists of two parts, namely part (a) and part (b). Part (a) models the arrival, queuing and service of the two types of events. In the figure, transitions $A1$ and $A2$ represent the arrivals of the events of types one and two, respectively. Places $B1$ and $B2$ represent the queue for the two types of events. Transitions $Sn1$ and $Sn2$ are immediate transitions which 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

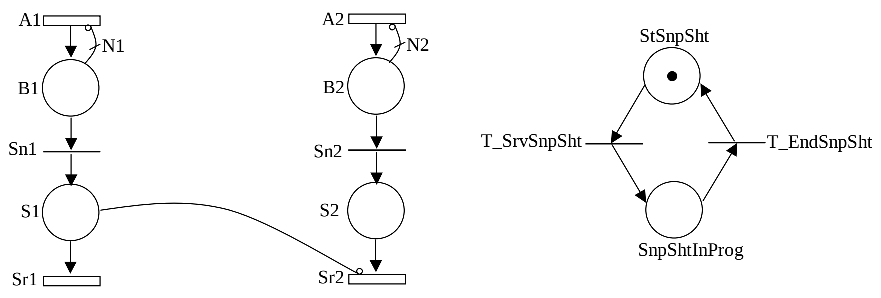

Figure 6: SRN model for the Reactor pattern

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 the place $B1$. The presence of $N1$ tokens in the 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 two 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 of one over events of type two.

Part (b) of the net models the process of taking successive snapshots and prioritized service of event handles corresponding to type one in each snapshot as explained below. 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. 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 place $S1$ 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 3 summarizes the enabling/guard functions for the transitions in the net.

We now explain how the process of taking a single snapshot is modeled by the SRN model presented in Figure 6 with an example. We consider 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, we assume that transition $Sn1$ fires before $Sn2$ depositing 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 $Sn2$. Since 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 $S2$. Once both the 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 4.2 can be computed by assigning reward rates at the net level as summarized in Table 4. The throughputs T_1 and T_2 are respectively given by the rate at which transitions $Sr1$ and $Sr2$ fire. The queue lengths Q_1 and Q_2 are given by the average 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 $B2$.

As described in Section 4.3, the waiting time for the two types of events needs to be determined using the tagged customer approach. The conditional waiting time for a tagged event of type one and type two 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

Table 3: Guard functions

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

Table 4: Reward assignments to obtain performance measures

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($\#B2 + \#S2$)
Response time of event type #1	R_1	return($\#B1 < N1?1/\mu_1 * (\#S1 + \#B1 + 1) + 1/\mu_2 * (\#S2 + \#B1) : 0$)
Response time of event type #2	R_2	return($\#B2 < N2?1/\mu_2 * (\#S2 + \#B2 + 1) + 1/\mu_1 * (\#S1 + \#B2 + 1) : 0$)

$S2$ determine the progress of the current snapshot, whereas, the markings of places $B1$ and $B2$ determine the state of the queue. The 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 either type can begin. Hence the time taken to complete the current snapshot contributes to the waiting time for each one of the event types. In order to obtain the entire waiting time of the two types of events, the contribution of the queued events of type one and type two needs to be determined.

Let n_1 be the number of events of type one in the queue, and n_2 be the number of events of type two in the queue, when the tagged event of type one 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 $n_1 < n_2$, then only n_1 of the type two events need to be serviced before the service of the tagged 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 one and type two need to be serviced before the service of the incoming type one 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 one and n_2 events of type two need to be serviced before the service of the tagged event can commence. The optimistic case assumes that no additional events of type two arrive in the first n_1 snapshots. In the pessimistic case, however, $n_1 - n_2$ events 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 pessimistic contribution of the queued events to the waiting time is given by the product of the number of tokens in place $B1$ and the sum of the reciprocals of μ_1 and μ_2. Thus, the overall response time of the tagged event will be given by the sum of two terms, the first term is $1/\mu_1$ times the sum of the tokens in places $S1$, $B1$ and 1, while the second term is given by the product of $1/\mu_2$ and the sum of the number of tokens in place $S2$ and $B1$. The reward rate to obtain the response time for an event of type one is summarized in Table 4.

The contribution of the queued events to the waiting time of the tagged event of type two can be determined as follows. Let n_1 be the number of events of type one in the queue, and n_2 be the number of events of type two in the queue, when the tagged type two event arrives. This implies that after n_2 snapshots the tagged event will be serviced. We now consider three cases similar to the tagged event of type one. When $n_2 < n_1$, the service time of the first n_2 snapshots is given by $n_2(1/\mu_1 + 1/\mu_2)$. In the $(n_2 + 1)^st$ snapshot in which the tagged event will be serviced, an event of type one will be serviced prior to the tagged event due to the priorities. Thus, the total waiting time will be given by $(n_2 + 1)/\mu_1 + n_2/\mu_2$. If $n_2 = n_1$, in the first n_2 snapshots, an additional event of type one may arrive, making the total waiting time as $(n_2 + 1)/\mu_1 + n_2/\mu_2$. If $n_2 > n_1$, then in the pessimistic case, in the first n_1 snapshots, an additional $n_1 - n_2 + 1$ events of type one arrive, making the total waiting time as $(n_2 + 1)/\mu_1 + n_2/\mu_2$. Thus, a pessimistic estimate of the total waiting time for given values of n_1 and n_2 is $(n_2 + 1)/\mu_1 + n_2/\mu_2$. Thus, the response time of the tagged event of second type will be given by the sum of two terms, the first term is $1/\mu_2$ times the sum of the tokens in places $S2$, $B2$ and 1, while the second term is given by the product of $1/\mu_1$ and the sum of the number of tokens in place $S1$, $B2$ and 1. The reward rate to obtain the response time for an event of type one is summarized in Table 4.

4.5 Illustration

In this section we illustrate how the SRN presented in Section 4.4 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 [4] to obtain the performance measures in all of the experiments described below.

In the first experiment we seek to determine the impact of maximum buffer capacity on the performance measures. For this experiment, the values of the remaining parameters (except for the buffer capacities) are summarized in Table 5. We consider two values of buffer capacities N_1 and N_2. In the first experiment we consider a buffer capacity of 1 for both types of events, whereas in the second experiment we consider a buffer capacity of 5 for both types of events. The performance metrics for both of the experiments are summarized in Table 6. Since 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 two E_2 is slightly higher than the total number of events of type one E_1. Since the events of type one are provided prioritized service over the events of type two, on an average it takes longer to service a type two event than it takes to service type one 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.

Table 5: Values of parameters

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

Table 6: Impact of buffer capacity on performance measures

Performance measure	Buffer space	
	$N_1 = 1, N_2 = 1$	$N_1 = 5, N_2 = 5$
T_1	0.37	0.40/sec
T_2	0.37	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
R_1	0.6298 sec.	0.8310 sec.
R_2	1.0973 sec.	1.3309 sec.

Next, we determine the sensitivity of the performance measures to the arrival rate of the events of type one and type two, that is, λ_1 and λ_2. For sensitivity analysis, we set the maximum buffer capacity for both the type of events to be 5. We vary both λ_1 and λ_2 one at a time in the range of 0.5/sec. to 2.0/sec. to obtain the different performance measures by solving the SRN shown in Figure 6. The remaining parameters are held at the values reported in Table 5.

Figure 7 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 into the system. 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 the type one 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 right plot in the top row) in Figure 7. 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 two events in the system increases as λ_1 increases. When λ_1 increases, the probability of having to service a type one event prior to servicing a type two event in a snapshot increases. Since events of type one have priority over events of type two, type two events tend to reside longer in the system as λ_1 increases. Thus, although the throughput of type two events is unchanged with respect to λ_1, the response time of a type two event may increase. The plots in the right column of Figure 7 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.

In the next experiment we determine the sensitivity of the performance measures to the service rates of the events, namely, μ_1 and μ_2. We set the maximum buffer capacity for both the type of events to be 5. We vary both μ_1 and μ_2 in the range of $1/2.5$/sec. to $1/0.5$/sec. in steps of $1/0.25$, one at a time to obtain the different performance measures by solving the SRN shown in Figure 6. The remaining parameters are held at the values reported in Table 5.

The plots in the left column of Figure 9 show the variation of the performance measures with respect to μ_1, while the plots in the right column of Figure 9 show the variation with respect to μ_2. The topmost plot in the left column shows the variation of throughputs of events of type one and two with respect to μ_1. The figure indicates that as μ_1 decreases, the throughputs of both types of events decreases, however, the magnitude of decrease in the throughput of the event of type one is more than the magnitude of decrease in the throughput of the event of type two. The second and the fourth plots in the same column shows the variation of the total number of events and the queue length of both types as a function of μ_1. It can be observed that both the total number of events and the queue length increases as μ_1 decreases, and the magnitude of increase for type one events is higher than the magnitude of increase for type two events. Similar trends can be observed in the variation of the loss probabilities and response times (plots third and fifth).

It can be noted that as μ_1 decreases, the performance measures deteriorate rapidly (loss probability, queue length, total number of events and response time increases, and throughput decreases). Also, as μ_1 decreases below 0.8/sec., the performance measures deteriorate rapidly. With decreasing μ_1, the time taken to service an event of type one increases, due to which the time taken to complete each snapshot increases. As a result, indirectly, this also increases the time taken to service an event of type two in each snapshot. Thus, the performance measures of both types of events are adversely impacted, even though only the service time of only one of the event types increases. This is in contrast to the variation of the performance measures as a function of the event arrival rates, where an increase in the arrival rate of a given type of event adversely impacts the performance measures of only that type of event.

The variation of the performance measures with respect to μ_2 shows the same trend, with the role of event types reversed.

5. DISCUSSION AND CONCLUDING REMARKS

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, often safety-critical distributed, performance-sensitive software (DPSS) systems.

The next-generation of DPSS systems, such as sensor network applications 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 DPSS 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 DPSS systems. This paper describes analytical performance evaluation techniques that help improve our understanding of the performance of the composed DPSS systems.

This paper concentrates on an important software building block called the Reactor, which is used in many DPSS systems. This paper shows that Stochastic Reward Nets (SRNs) provide a powerful analytical technique to model building blocks, such as the Reactor. Moreover, we show that SRNs provide the desired capabilities to evaluate different performance metrics for DPSS systems, including throughput, event loss, queueing delays, and even sensitivity analysis. It is important to apply these metrics as early as possible in the DPSS system lifecycle (*e.g.*, design-time) since DPSS developers can use these results to guide decisions regarding appropriate resource provisioning and schedulability analysis.

The primary goal of this paper was to evaluate the applicability of different analytical techniques to accurately model and evaluate individual software building blocks. The insights we have gained will shape our future R&D efforts in this area, which involve developing analytical models for larger DPSS systems that are composed out of building blocks. This in turn will enable us to evaluate performance of entire DPSS systems that are assembled out of different software building blocks.

Our future work will also validate the performance evaluations of our analytical models via empirical benchmarking. We omit these results in this paper due to the complexities 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. We plan to demonstrate these capabilities once our models for composable building blocks have evolved. We also expect to provide results for more complex versions of the Reactor, including the *thread pool* Reactor [16].

6. REFERENCES

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, New York, NY, 1977.

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

[3] B. R. Haverkort and K. S. Trivedi. “Specification and generation of Markov reward models”. *Discrete–Event Dynamic Systems: Theory and Applications*, 3:219–247, 1993.

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

[5] 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.

[6] 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.

[7] 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.

[8] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. *Proceedings of the IEEE*, 91(1):145–164, Jan. 2003.

[9] 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.

[10] Object Management Group. *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 edition, July 2001.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[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.

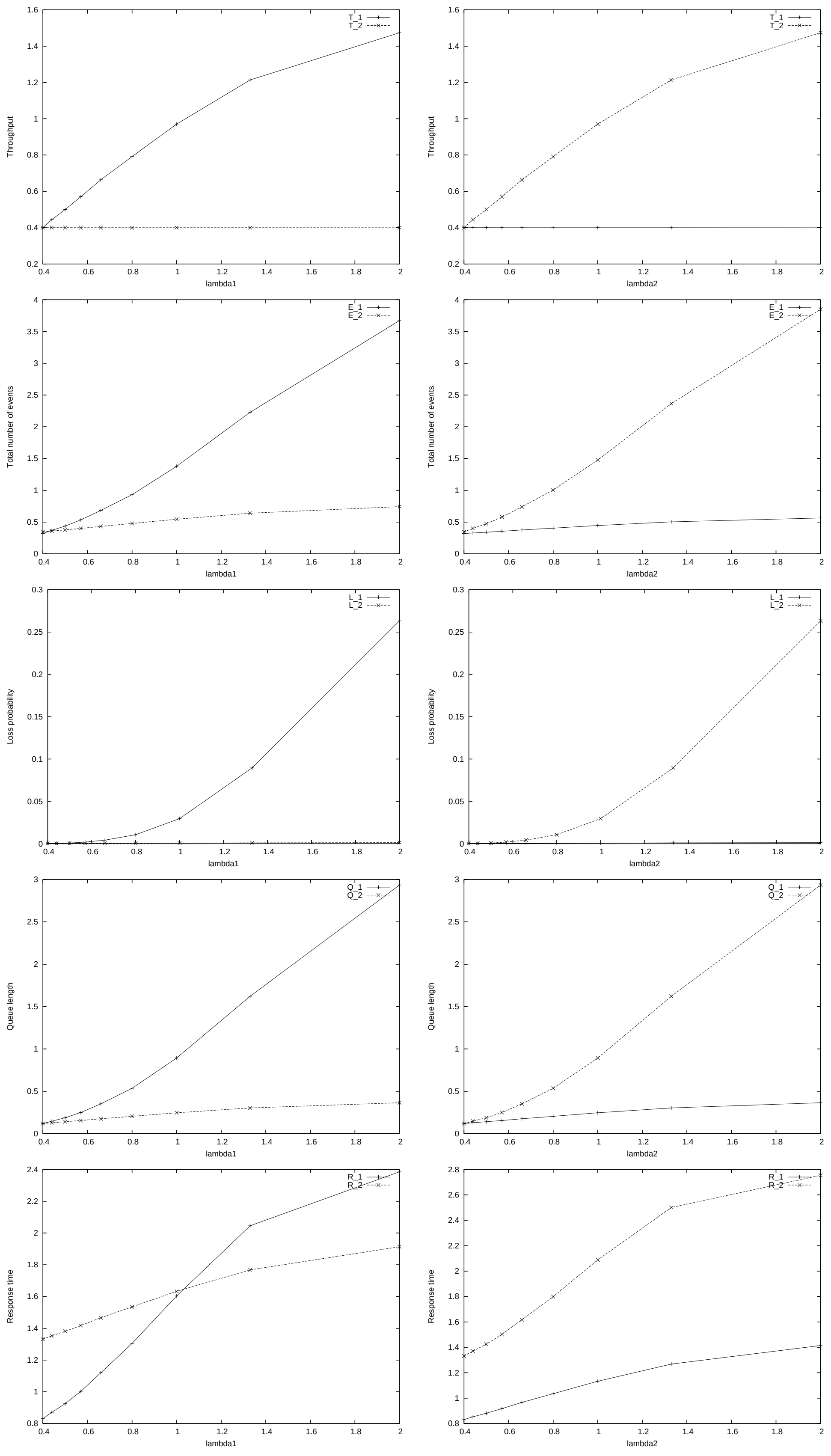

Figure 7: Sensitivity of performance measures to arrival rates λ_1 and λ_2

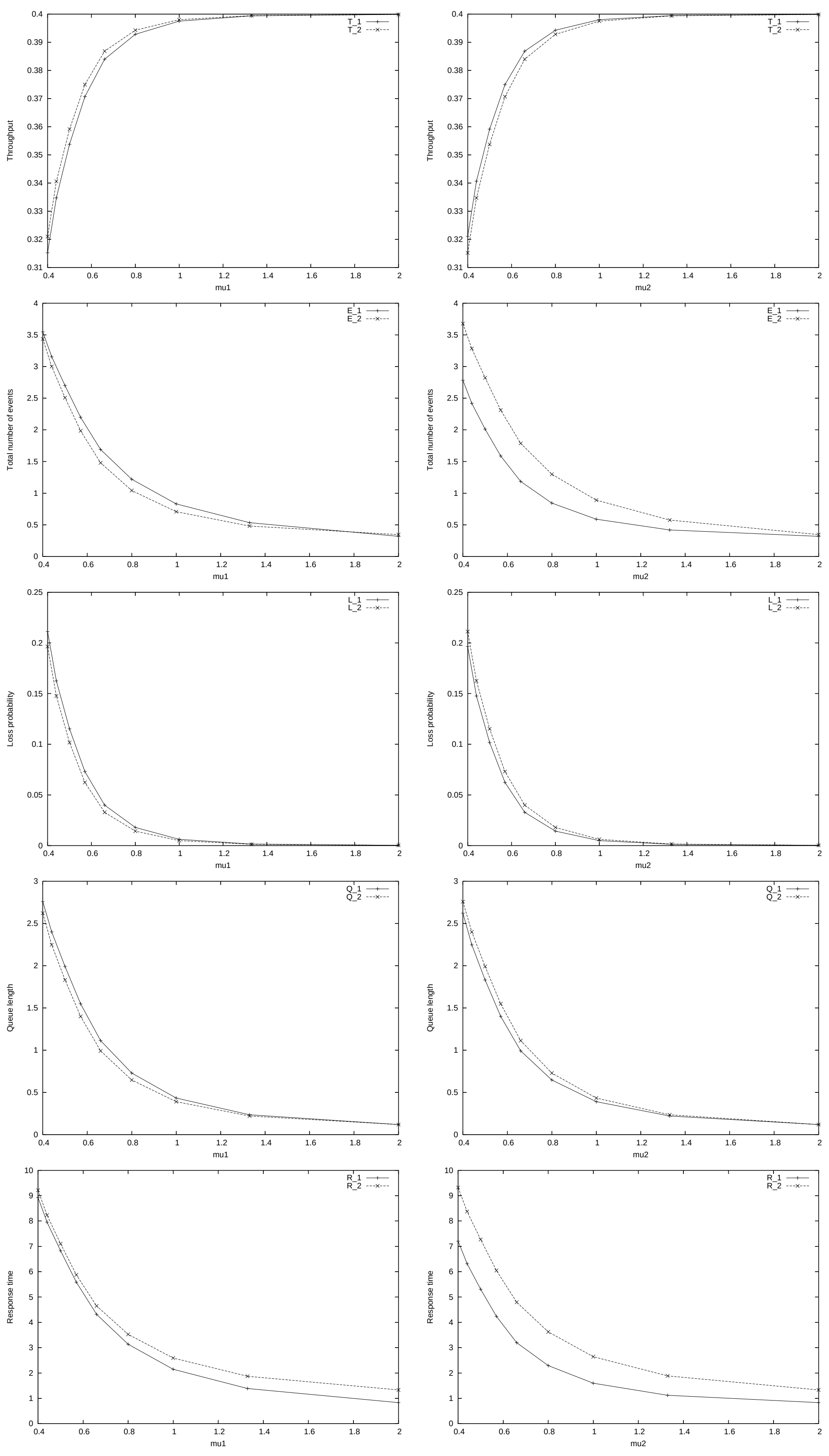

Figure 8: Sensitivity of performance measures to service rate μ_1 and service rate μ_2

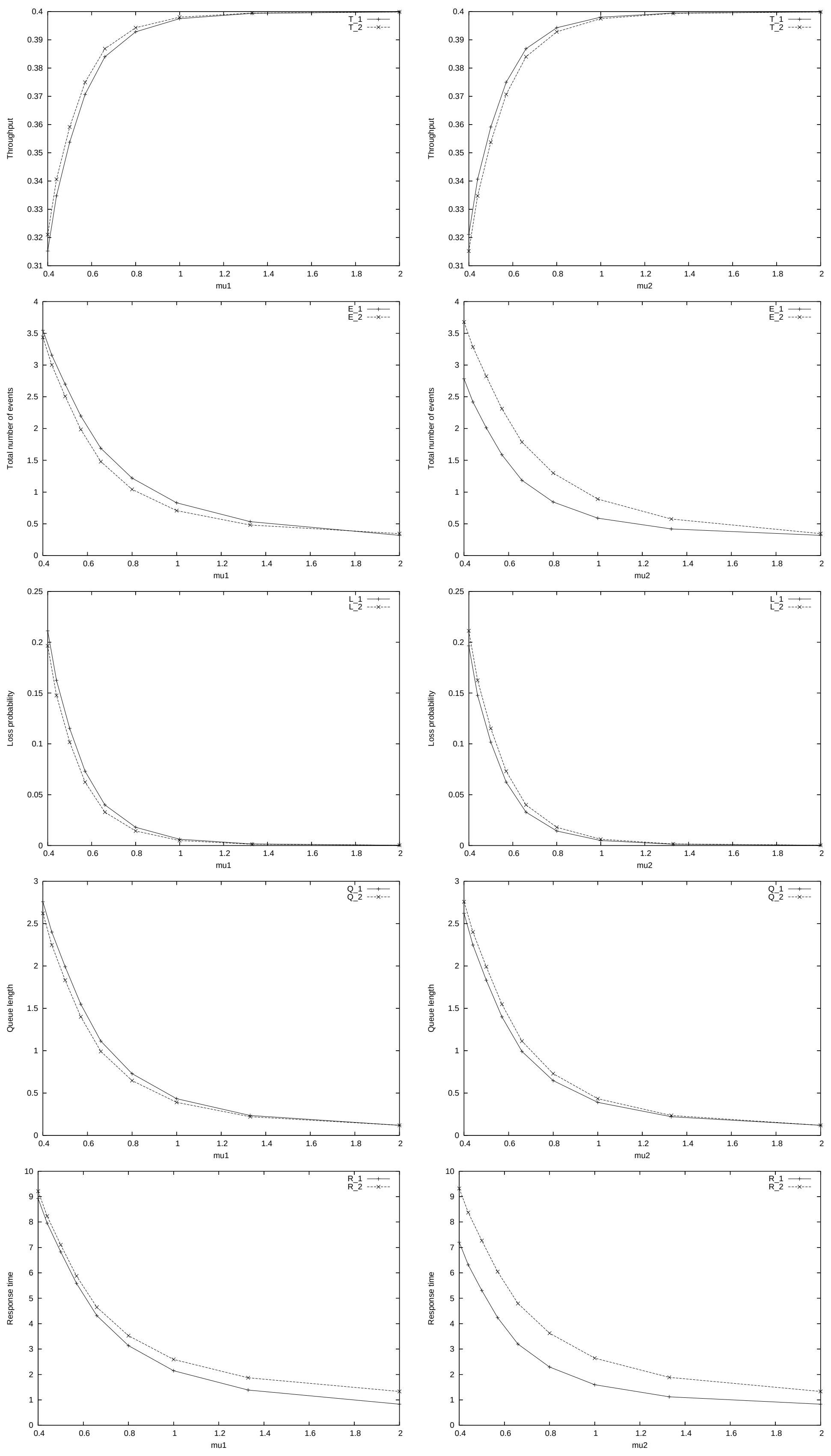

Figure 9: Sensitivity of performance measures to service rates μ_1 and μ_2