Performance Evaluation of the Reactor Pattern Using the OMNeT++ Simulator

Arundhati Kogekar, Aniruddha Gokhale
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, Tennessee 37235
{akogekar, gokhale}@dre.vanderbilt.edu

ABSTRACT

The design of large-scale, distributed, performance-sensitive systems presents numerous challenges due to their network-centric nature and stringent quality of service (QoS) requirements. Standardized middleware implementations provide the key building blocks necessary to address these requirements of the distributed systems. However, middleware are designed to be applicable for a wide range of domains and applications, which results in system developers requiring to choose the right set of building blocks to design their system. To reduce the impact on development costs and time-to-market, decisions on the right set of building blocks to use in systems design must be made as early as possible in system design. This paper addresses this concern by describing a model-driven systems simulation approach to analyze, catch and rectify incorrect system design decisions at design-time. In this paper we focus on model-driven OMNeT++ simulation of the Reactor pattern, which provides event demultiplexing and handling capability. Our experience with modeling the Reactor shows that this approach can be extended to the performance analysis of other pattern-based blocks and indeed in the long term to the entire composed middleware framework.

Categories and Subject Descriptors

I.6.5 [**Simulation and Modeling**]: Model Development

General Terms

Performance Analysis

Keywords

Distributed systems, middleware, performance modeling

1. INTRODUCTION

Standardized middleware, such as J2EE, CORBA and .NET, is used in building real-time, performance-critical distributed systems in many domains, such as telecommunications, power grid and enterprise systems among others. Middleware provides the mechanisms to ensure scalability, fault tolerance, reliability and other quality of service (QoS) requirements of these distributed systems. Since standardized middleware implementations are designed to be applicable to a wide range of domains and applications they come equipped with a number of building blocks with customizable configurations. It is then the task of the system developer to select the right building blocks, evaluate the performance of these configurations and select the one which are best suited for that particular domain. In traditional systems development, the developer often has to wait very late into the system lifecyle, for example, until runtime, to benchmark various configurations, which is both costly and time-consuming. If the configuration which provides the most optimum performance is known at design time, code can be written only for that configuration. Similarly, the sooner the design flaws are detected in the system lifecycle, the easier and cheaper it is to correct them. This highlights the benefit of using design-time performance analysis, such as simulations, to predict the system performance at design time, when there is still time and ample opportunity to change the design, without causing additional wastage of resources.

The design of contemporary standardized middleware is based on elegant patterns as well as pattern languages [1]. Patterns-based middleware is a class of middleware, which is designed and implemented by composing together different pattern-based functional building blocks. Patterns [1] represent solutions to a common set of problems arising in a particular context. A pattern therefore is a body of expert knowledge on best practices, designs and strategies that has been documented in a standardized manner and that can therefore be reused in similar situations. In the context of middleware, patterns represent solutions for common distributed and network programing tasks such as event handling, memory management, service access and configuration, concurrency and synchronization [7]. Figure 1 [8] shows a pattern-based middleware architecture composed of building blocks representing these patterns.

The emphasis we laid earlier on design-time performance analysis of distributed systems now maps to the performance evaluation of the patterns-based middleware building blocks. For this paper we are interested in using simulation as the mechanism for design-time performance evaluation. The simulation of a pattern-based middleware building block, however, presents its own set of challenges. Each middle-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SE'06 March 10-12, 2006, Melbourne, Florida, USA
Copyright 2006 ACM 1-59593-315-8/06/0004 ...$5.00.

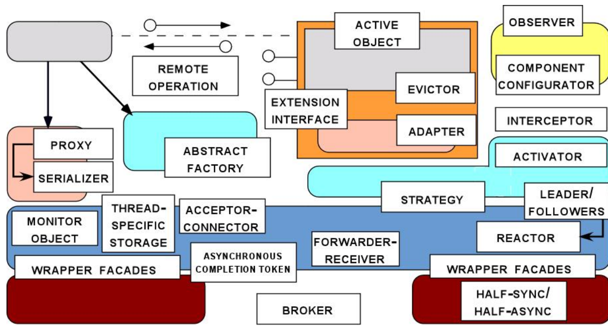


Figure 1: Patterns in Middleware

ware block interacts in different ways with other blocks. For example, the reactor pattern has to deal with numerous simultaneous events. The event-handling mechanism in the reactor allows concurrency by enabling event handling in multiple threads concurrently. This necessitates the use of a powerful discrete event-based simulator which would be able to simulate the simultaneous occurrence and handling of events. In addition, the simulator must be able to incorporate any modifications to the model easily, i.e, the simulation of the combination of two or more building blocks should not require extensive rewriting of existing simulation code. We have chosen the OMNeT++ (`www.omnetpp.org`) simulator to model and evaluate the middleware blocks.

The organization of the rest of the paper is as follows: Section 2 provides an overview of the structural and dynamic aspects of the Reactor building block, which we have simulated for this paper. Section 3 provides an overview of OMNeT++ and describes the simulation model of the Reactor pattern. Section 4 explains the simulation set up and presents an analysis of the results obtained from this simulation. Section 5 discusses some of the related research in this area. Finally, Section 6 concludes the paper by explaining the future direction of this approach and providing some insights and lessons learned in the course of designing this simulation model.

2. THE REACTOR PATTERN

The ability to handle and dispatch simultaneously occurring events effectively without any additional resource overhead is an integral part of middleware used in a real-time, event-driven and performance-critical environment. For correct functionality of the entire system, the performance of middleware for this particular task needs to be optimum and hence must be evaluated carefully. This paper focuses on the performance evaluation of the "Reactor Pattern", which is at the heart of the event handling mechanisms in middleware.

The Reactor architectural pattern [7] allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients. The Reactor pattern inverts the flow of control in a system during event handling. Figure 2 [7] shows the structural composition of the reactor pattern.

The Reactor pattern consists of the following participants:

- *Handle*: The handle uniquely identifies event sources such as network connections or open files. Whenever an event is generated by an event source, it is queued up on the handle for that source and marked as "ready".

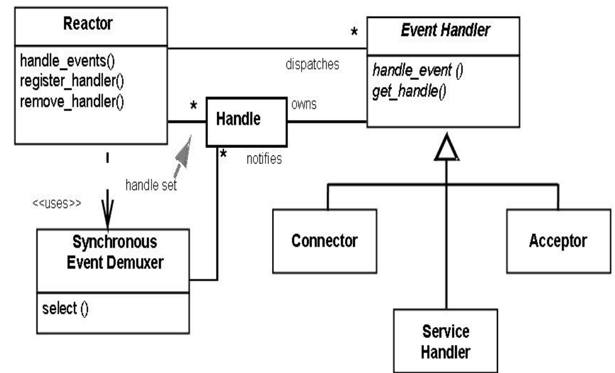


Figure 2: Structural Represenation of the Reactor Pattern

- *Reactor*: The reactor is the dispatching mechanism of the Reactor pattern. In response to an event, it dispatches the corresponding event-handler for that event.
- *Event Handler*: The event handlers are the entities which actually process the event. These are registered with the reactor and are dispatched by the reactor when the event for which they are registered occurs. The handles for which these handlers are registered form a "Handle Set".
- *Synchronous Event Demultiplexer*: This entity is actually implemented as a function call, such as `select()` or `WaitForMultipleObjects()` (in case of Windows-based systems). It waits for one or more indication events to occur, and then propagates these events to the reactor.
- *Concrete Event Handlers*: The concrete event handlers specialize the generalized Event Handler. They are responsible for processing specific types of events. Figure 2 shows three types of concrete event handlers: the Acceptor, the Connector and the Service Handler.

Figure 3 [7] illustrates the dynamics of the Reactor pattern.

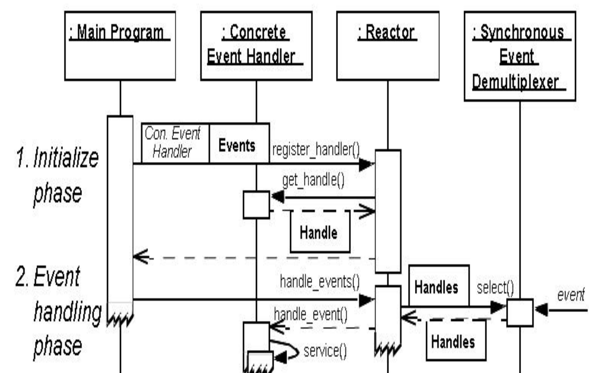


Figure 3: Dynamics of the Reactor Pattern

In the initialization phase, the application registers its event handlers, as well as the handles that they are prepared to accept, with the reactor. The reactor collects all the registered handles into a handle set. In the event handling phase, whenever an event occurs, the synchronous event demultiplexer channels it to the reactor. Based on the handle for that event, the reactor dispatches the corresponding registered event handler. The event handler then processes the event. In this way, the application does not spend its time polling for an event to take place, nor does it have to deal with demultiplexing the event to the appropriate handler. The reactor is present to do this job. Additional information about the Reactor architectural pattern can be found in [7].

3. MODELING THE REACTOR SIMULATION

This section describes the simulation model of the reactor pattern. After careful study we chose OMNeT++ (`www.omnetpp.org`) as the simulator for our research because of its ease of use, flexible and modular architecture, parametric approach and open-source code base. OMNeT++ also has an advantage over other existing simulators in that it easily allows for the simulation of virtually any modular, event-driven system, and not just communication-network oriented systems.

OMNeT++ [10] follows a hierarchical architecture. At the lowest level of the hierarchy are simple modules which encapsulate behavior. These simple modules are represented by C++ classes. A compound module may be composed of simple as well as other compound modules. Modules communicate with each other via message-passing. An event is said to have occurred whenever a module sends/receives a message. A module may have parameters whose values are specified externally in an initialization file. These parameters can be varied in different simulation runs. In the context of middleware, these parameters can be used to simulate and analyze the effect of different middleware configuration options. Additional information about OMNeT++ can be found in the OMNeT++ User Manual [9].

The Simulation Model

The simulation model for the Reactor pattern is based on the structure of the Reactor as shown in Figure 2. The topology of the model is shown in Figure 4. This topology is specified in the .NED file of OMNeT++.

Our simulation model consists of the following blocks:

- *Event Generators*: Event generators are event sources, which generate events at a Poisson distribution rate λ. The number of event generators and their rates of event generation are parameterized values.

- *Synchronous Event Demultiplexer*: The synchronous event demultiplexer receives the events generated by the Generators. Depending on which generator generated the event, the synchronous event demultiplexer attaches an *Event_Type* value to the event and subsequently propagates the event to the Reactor.

- *Reactor*: Depending on the *Event_Type*, the reactor dispatches and activates the appropriate Event Handler by sending an event to that handler.

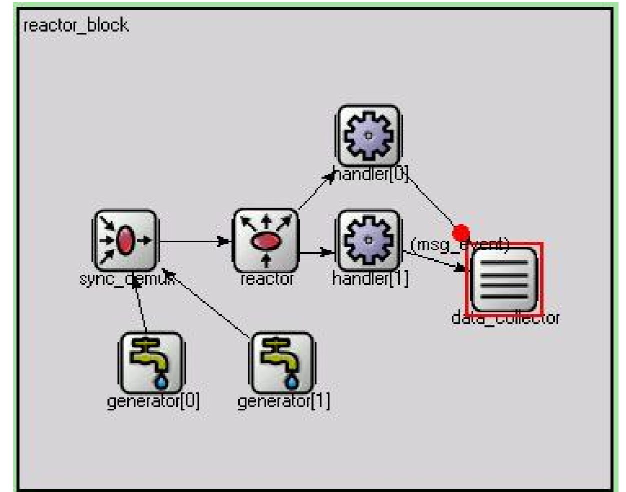


Figure 4: Simulation Model of Reactor

- *Event Handlers*: Each Event Handler has an exponentially distributed service time with rate μ. Each Event Handler also has a bounded queue associated with it with a maximum size of N. Upon receiving the dispatch event from the Reactor, the event is immediately handled if the queue is empty and no other event is being handled. If an event is currently being handled and the queue is not full, the incoming event is queued. If the queue is full, then the event is dropped. After an event has been handled, the event handler propagates it to the data collector. The event-handling process is simulated by scheduling the event to be propagated after a delay of *Service_Time* seconds. The number of event handlers as well as the service rate μ of each handler is a parameterized value and can be changed for each simulation run.

- *Data Collector*: The data collector acts as an event sink. It receives events sent by the Event Handlers. The data collector also calculates the throughput value and loss probability for each Event type.

We have modeled the generators as generating events at a Poisson distribution because the generated events represent the arrival pattern of events into the system, which is most commonly taken to be Poisson. Similarly, the service times of Event Handlers are exponentially distributed according to the most common service pattern. We have modeled a bounded buffer for Event Handlers as most of the real-time, event-driven systems do not have the memory resources required for an infinite buffer.

Statistics Collection

The following metrics [3] are measured during the simulation process:

- *Throughput(T)*: The throughput for each event type i is calculated by the data collector as the number of events of that type received by the collector divided by the simulation time at the end of the simulation run. The throughput metric is important for real-time event processing and distributed applications, such as online stock trading services.

- *Queue Length (Q)*: The queue length for each event type is recorded each time an event arrives for the event handler for that type. The queue length metric is significant for resource-constrained systems, such as RFID chips, that need to know the optimum buffer size to allocate for buffering events.
- *Loss Probability (L)*: The loss probability for an event type i is calculated by the data collector as the number of events sent by the event handler divided by the total number of events arriving in the event handler. This metric is significant for hard real-time systems where the loss of a control event would significantly affect the performance and even correctness of the system.

In the next section we describe the actual simulation set up and present an analysis of the results.

4. SIMULATION RESULTS AND ANALYSIS

This section describes the results of simulating the reactor pattern in OMNeT++ by varying different parameters. The number of event generators, as well as the number of event handlers, is set to two. Table 1 lists out the input parameters, the performance metrics and their notations.

(a) Parameters

Parameter	Type 0	Type 1
Arrival Rate	λ_0	λ_1
Service Rate	μ_0	μ_1
Maximum Buffer Length	N_0	N_1

(b) Metrics

Metric	Type 0	Type 1
Mean Queue Length	Q_0	Q_1
Throughput	T_0	T_1
Loss Probability	L_0	L_1

Table 1: Notations

The initial values of the input parameters are shown in Table 2.

Parameter	Initial Value
λ_0	0.4/s
λ_1	0.4/s
μ_0	2.0/s
μ_1	2.0/s
N_0	5
N_1	5

Table 2: Initial Set-Up

The input parameters are specified in the *omnetpp.ini* file. These parameters are read at runtime by the *OMNeT++* simulation environment for each set of simulation runs. A sample *omnetpp.ini* file used for analyzing the effect of the arrival rate λ_0 is given below:

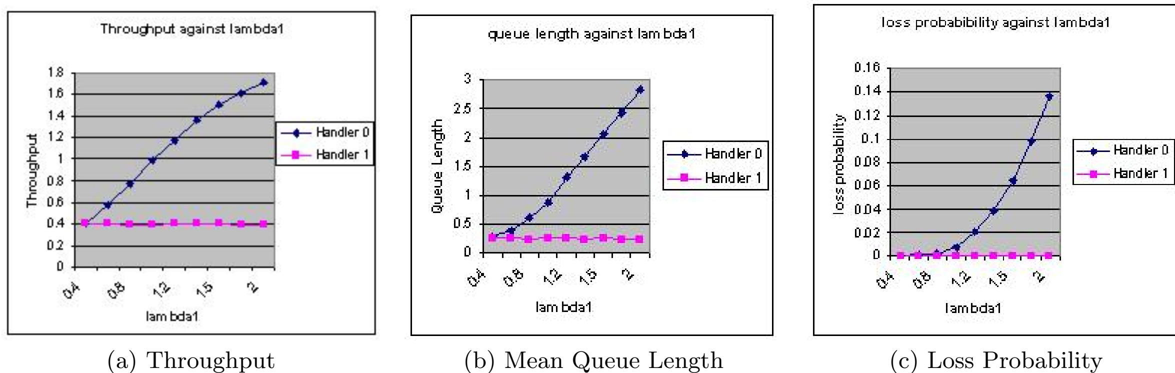
```
[General]
preload-ned-files=*.ned
network=reactor_block
sim-time-limit=10000s

[Parameters]
reactor_block.num_handlers=2;
reactor_block.generator[1].lambda=0.4;
reactor_block.handler[0].mu=2;
reactor_block.handler[1].mu=2;
reactor_block.handler[0].queue_size=5;
reactor_block.handler[1].queue_size=5;

[Run 0]
reactor_block.generator[0].lambda=0.4;
[Run 1]
reactor_block.generator[0].lambda=0.6;
[Run 2]
reactor_block.generator[0].lambda=0.8;
[Run 3]
reactor_block.generator[0].lambda=1.0;
[Run 4]
reactor_block.generator[0].lambda=1.2;
[Run 5]
reactor_block.generator[0].lambda=1.4;
[Run 6]
reactor_block.generator[0].lambda=1.6;
[Run 7]
reactor_block.generator[0].lambda=1.8;
[Run 8]
reactor_block.generator[0].lambda=2.0;
```

Effect of Arrival Rate. For the first set of simulation runs, the effect of the arrival rate λ_0 on the throughput, mean queue length and probability of event loss was measured. As noted in the sample *omnetpp.ini* file, λ_0 was varied from 0.4 to 2.0 in steps of 0.2, while the other input parameters were kept constant at the values given in Table 2. The results are shown in Figure 5. It can be seen that as the arrival rate for Event Type 0 increases, the throughput for Type 0 also increases. The throughput for Type 1 remains constant, since arrival and processing of Type 0 is independent from Type 1.

It can also be seen that as the arrival rate increases, the loss probability of Type 0 events increases, i.e, more Type 0 events are likely to be dropped. This can also be correlated to the increase in the mean queue length of Type 0 events.

Effect of Service Time. For the second set of simulation runs, the effect of the service rate μ_0 on the throughput, mean queue length and probability of event loss was measured. This time μ_0 was varied from 0.4/s to 2.0/s in steps of 0.2, while other input parameters were kept constant at the values given in Table 2. The results are shown in Figure 6. It can be seen that as μ_0 increases (i.e the time required by Handler 0 to process the events decreases) the throughput for Type 0 increases. The throughput for Type 1 remains constant, since arrival and processing of Type 0 is independent from that of Type 1.

It can also be seen that the loss probabibility of Type 0 events decreases as service time decreases, since the number of queued events decrease with decrease in service time. This can also be deduced by the decrease in Mean Queue Length as seen in Figure 6(b). It can be seen from Figure 6(c)



(a) Throughput (b) Mean Queue Length (c) Loss Probability

Figure 5: Effect of Arrival Rate

that the probability of loss increases rapidly when μ_0 drops below 0.8/s. This would be useful information for a system developer who needs to know the maximum allowable service time for a given loss probability.

It should be noted that the simulation model of the Event Handler does not take into account *how* the handler actually handles the event. The simulation therefore does not consider the effects of implementation artifacts such as the internal data structures used. For the purposes of simulation, the handler is considered to be a *black box*.

Effect of Maximum Buffer Size. For the third set of simulation runs, the effect of the maximum buffer sizes N_0 and N_1 on the throughput, mean queue length and probability of event loss was measured. Other input parameters were kept constant at the values shown in Table 2. N_0 and N_1 were both kept at 1 for the first run. The results are illustrated in Figure 7. It can be seen that for a maximum buffer size of 1, all three metrics, i.e. throughput, mean queue length and loss probability were sub-optimal. In the second run, N_0 and N_1 were 5. The change in the results was remarkable, especially for the loss probability, as seen in Figure 7(c). The loss probability was almost zero for buffer size of 5. This indicates that the system was able to sustain the given arrival pattern. It also serves as a useful indicator to resource provisioners about the capacity of the system to handle higher event arrival rates for given buffer constraints.

The throughput also increased, since considerably less number of events were being dropped. As expected, the mean queue length also increased with increase in maximum buffer size.

These results provide some insightful information about the design of the system. For example, from Figure 5 it can be seen that with increase in the arrival rate there is increase in the throughput, but the loss probability is increased as well. Therefore for higher arrival rates the designer could provide multi-level queues in his/her design to minimize the probability of loss. Also, by studying the effect of a combination of parameters such as maximum buffer size and the service rate, the designer could decide the most optimum configuration for the system.

While developing this simulation model manually, we observed that the scalability of the model stood out as an important issue. We found that as the number of event handlers and event generators increases, it becomes quite difficult to construct a correct simulation model by hand. Similarly, if we add a few more patterns such as Acceptor-Connector to the simulation model, it will be extremely difficult to manage the entire model manually. We are therefore exploring the use of model-driven generative techniques for generating simulation models auotmatically. These techniques would factor out some of the common tasks in simulation (such as adding new connections upon addition of a new handler). They could also guard against any errors introduced by changes to the model. The use of model-driven techniques in this context is elaborated in Section 6.

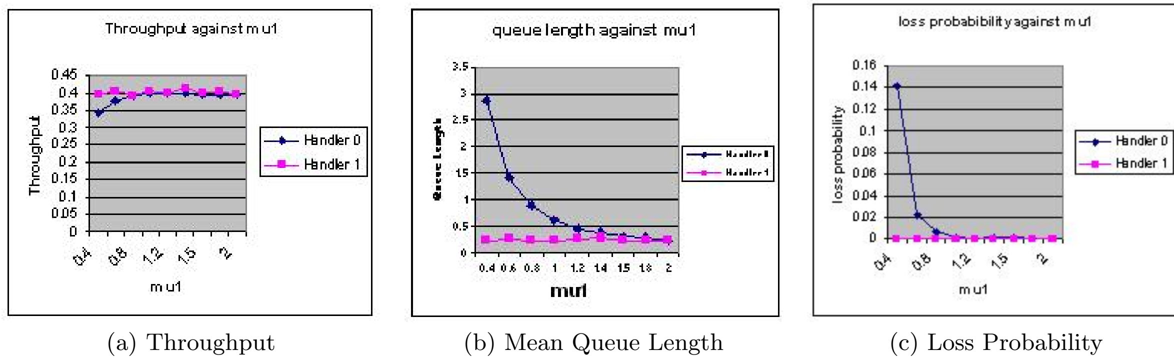
5. RELATED WORK

There have been efforts to evaluate the performance of middleware patterns analytically by various researchers [2], [6]. A drawback of using analytical models is that it is difficult to predict the behavior of a complex system based on analytical methods alone. Harkema, *et al* [4] have worked on the performance evaluation of the CORBA method invocation and threading models. However, they have not focused on the pattern-based approach toward the performance analysis of middleware. Model-driven techniques are increasingly being used for middleware development, but converting static pattern-based middleware models into simulation models for the purpose of performance evaluation has not yet been a focus of research in the research community.

6. FUTURE WORK AND CONCLUSIONS

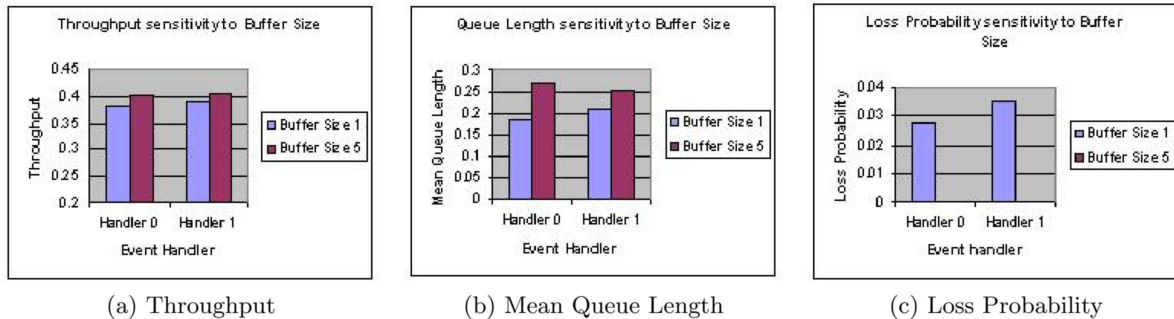
In conclusion, the simulation model of the reactor pattern represents the first step in our bottom-up approach toward analysis of pattern-based middleware. It provided us some insight into the event-handling behavior of middleware. This experience should prove useful in the simulation and analysis of other building blocks as well as of the composed system. Thus, our approach illustrates a mechanism for design-time performance analysis of middleware building blocks that are required to develop large, distributed and performance-critical systems.

As discussed earlier, middleware is built of pattern-based building blocks such as Acceptor-Connector, Proactor, Broker, etc [7]. By modeling these patterns and their inter-pattern and intra-pattern dynamics, it will be possible to develop a simulation model for a composed pattern-based middleware system that is tailored to the needs of the ap-



(a) Throughput (b) Mean Queue Length (c) Loss Probability

Figure 6: Effect of Service Time



(a) Throughput (b) Mean Queue Length (c) Loss Probability

Figure 7: Effect of Maximum Buffer Size

plications. Thus, our future work in this area will focus on enhancing this simulation model as well as building simulation models of other patterns. Another research goal is to map a structural model designed in a modeling environment, such as the Generic Modeling Environment(GME) [5] to the simulation model in OMNeT++, so that changes in the GME model would automatically be reflected in the simulation model or any other back-end performance analysis models. The various configurations modeled in GME could then be simulated and evaluated in OMNeT++ at the "click of a button". This will be helpful to developers who are using the model-driven development approach and who want to evaluate the performance of their design by simulation.

7. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison Wesley Professional, 1994.

[2] S. Gokhale, A. Gokhale, and J. Gray. Model-driven performance analysis methodology for distributed performance sensitive software systems, 2005.

[3] D. Gross. *Fundamentals of Queueing Theory*. Wiley Series in Probability and Statistics. Wiley-Interscience, 3 edition, 1998.

[4] M. Harkema, B. M. M. Gijsen, R. D. van der Mei, and Y. Hoekstra. Middleware performance: A quantitaive modeling approach. In *International Symposium on Performance Evaluation of Computer and Communication Systems (SPECTS)*, 2004.

[5] A. Ledeczi, B. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, S. J., and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 2001.

[6] 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.

[7] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Addison Wesley, 2000.

[8] D. C. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: their synergistic relationships. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 694–704, 2003.

[9] A. Varga. *The OMNeT++ User Manual*, 1997.

[10] A. Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, 2001.