Performance Analysis of the Active Object Pattern in Middleware

Paul J. Vandal, Swapna S. Gokhale
Dept. of CSE
Univ. of Connecticut
Storrs, CT 06269
{pvandal,ssg}@engr.uconn.edu

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

Abstract

A number of enterprises are turning towards the Service Oriented Architecture (SOA) approach for their systems due to the number of benefits it offers. A key enabling technology for the SOA-based approach is middleware, which comprises of reusable building blocks based on design patterns. These building blocks can be configured in numerous ways and the configuration options of a pattern can have a profound impact on system performance. A performance analysis methodology which can be used to assess this influence at design time can guide the selection of patterns and their configuration options and thus alleviate the possibility of performance problems arising later in the life cycle.

This paper presents a model-based performance analysis methodology for a system built using the Active Object (AO) pattern. The AO pattern is chosen because it lies at the heart of an important class of producer/consumer and publish/subscribe systems. Central to the methodology is a queuing model which captures the internal architecture of an AO-based system. Using an implementation of the queuing model in CSIM, we illustrate the value of the methodology to guide the selection of configuration and provisioning options for a stock broker system.

1 Introduction and motivation

The introduction of distributed components into the process of Enterprise Application Integration (EAI) has moved traditional integrations towards a more Service Oriented Architecture (SOA) based approach [6]. The SOA-based approach offers advantages such as robust, scalable, and cost-effective systems, achieved by reducing complexity and eliminating redundant code. Since these systems will be used in many critical domains, they will be expected to satisfy multiple Quality of Service (QoS) attributes.

A key enabling technology for SOA-based systems is QoS-enabled middleware [9], which comprises of building blocks based on design patterns to codify solutions to the commonly recurring problems. These patterns are highly flexible since they allow a system to be customized as per its requirements through an appropriate selection of configuration options. The configuration options of a pattern, however, exert a strong influence on system performance. Despite this influence, current trend in the performance analysis of these systems relies on empirical benchmarking and profiling, which involves measuring the system performance after it is implemented. These types of testing techniques, which are applicable very late in the life cycle, can be detrimental to the cost and schedule of a project, since several design and implementation iterations may be needed to achieve the expected performance. A systematic methodology to facilitate design-time performance analysis can guide the process of selecting patterns and their configuration options and may thus alleviate these pitfalls.

In this paper we present a model-based performance analysis methodology for a system built using the Active Object (AO) pattern [10, 5]. The AO pattern is chosen since it is widely used in a class of producer/consumer and publish/subscribe systems. At the heart of the methodology is a queuing model that captures the internal architecture of an AO-based system. Using a CSIM implementation of the queuing model [11], we illustrate the value of the methodology in guiding configuration and provisioning decisions for a case study of a stock broker system.

The paper is organized as follows: Section 2 provides an overview of the AO pattern. Section 3 presents the methodology. Section 4 illustrates the methodology with a case study. An overview of related research is in Section 5. Concluding remarks and future directions are in Section 6.

2 Description of the AO pattern

In a multi-threaded application, several threads may require the utilization of a common resource. These threads then compete for mutually exclusive access to the resource and utilize it for the total time taken to complete the required operation. For low request rates and short session durations, the performance of this architecture may be acceptable. However, for high request rates and long access times, performance degradation may be significant. The AO pattern can be used to alleviate the performance problems in such a system. This pattern provides concurrency and simplifies synchronized access to the shared resource by decoupling method invocation from method execution and creating the shared resource in its own thread of control.

The AO [9] is composed of the following components: Proxy, Activation List, Scheduler, Servant, and Method Requests. The interactions between these are initiated by a client thread invoking a method on the Proxy to the AO. The Proxy lies in the client thread and provides an interface to the public methods on the shared resource. Instead of immediately executing the method when invoked by the client thread, the Proxy constructs a Method Request and enqueues it on the Activation List of the AO. Thus, from the client thread's perspective, the method has been executed.

The Method Request is a structure that carries the parameters along with the other information necessary to execute the request later. It also has guards or synchronization constraints. The Activation List is a buffer which resides in the thread of the AO and holds all the pending requests. A Scheduler monitors the Activation List for requests that meet their synchronization constraints. It then chooses a request to be executed, dequeues it, and dispatches it to the servant, which actually executes the method.

The AO pattern can be used to implement a class of publish/subscribe and producer/consumer systems. In this paper, we focus on an AO-based producer/consumer system.

3 Performance analysis methodology

In this section we discuss the performance analysis methodology for an AO-based producer/consumer system. First, we describe the characteristics of a mutex-based producer/consumer system, which is then enhanced through the use of the AO pattern to mitigate its performance problems. We then present queuing models of mutex- and AO-based systems, followed by a discussion of the metrics that can be used to gauge system performance. Finally, we describe the implementation of the queuing models in CSIM.

3.1 System characteristics

We consider a producer/consumer system in which two applications act as producers to a remote consumer application. In such a system, the producers and the consumers require access to a common resource, for example, a message buffer. The system thus requires a synchronization strategy to create thread safe access to the resource.

Figure 1 shows a system implementation in which mutex constraints are used for multi-threaded synchronization. The solution comprises of a Consumer Handler which exists in its own thread of control and serves as a proxy to the consumer application. This handler contains a Message Queue for outgoing messages that is implemented with the Monitor Object pattern [10] to allow thread-safe synchronous access to the queue. It also contains a Message Broker that is responsible for monitoring the queue for new messages to be sent to the consumer. When the Message Queue contains messages, the Message Broker will contend with the producers to access it. Once it gains access, the Message Broker will get a message from the queue and send it to the consumer application. Additionally, the two producers contend for access to the Message Queue to put messages into it. When the Message Broker is actively working on the get and send functions, the Message Queue is locked from access. Similarly, the Message Queue is locked when a producer is trying to put a message on it. Thus, once an entity (a producer or the Message Broker) acquires the mutex lock from the Monitor Object, it retains control of the Message Queue until its transaction is complete, after which it releases the lock. Thus, the duration of these access times is defined by network latency. For low to moderate network loads, these access times are short and the system performance may be acceptable. In a congested network, however, long access times, partly driven by the TCP flow control, may cause performance problems and starvation of the entities from accessing the Message Queue.

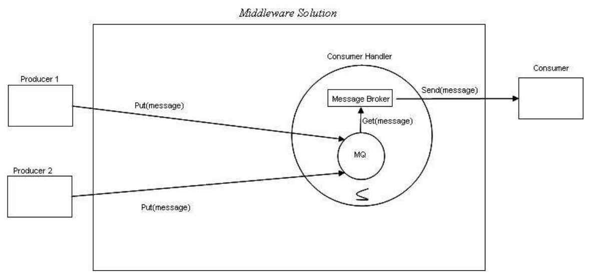

Figure 1. Mutex-based system

The above issues of the mutex-based system can be alleviated by using the AO pattern to decouple producers and consumers as shown in Figure 2. To decouple a producer, a Producer Handler Proxy to the Consumer Handler is introduced and implemented as a distributed AO. Its purpose is to receive messages from the producer and then put them in the Consumer Handler's Message Queue. The AO Proxy resides on the client application and provides an interface for the method to put messages on the Consumer Handler's Message Queue. When the put command is invoked by the client, the Proxy creates the corresponding Method Request and enqueues it on the Producer Handler's Activation List. The synchronization constraint of the put request is the requirement of the Proxy to gain control of the Message Queue. When the synchronization constraint is satisfied, the Scheduler dequeues the request and executes the method to put the message on the Message Queue. Thus, the time required to add a message to the queue is reduced to the internal access time of the middleware, which decouples the impact of the network latency on the producer side.

To decouple the consumer, the sending mechanism of the Message Broker is also implemented using an AO. A proxy interface containing the send method is implemented inside the Message Broker. When the Message Broker invokes the method to send a message, a Method Request is created by the Proxy and enqueued on the Activation List of the consumer-side AO. From the Message Broker's perspective, the sending of the message is nearly instantaneous, allowing it to relinquish control of the Message Queue after the small time required to get the message and to invoke the send command. This allows the affect of network latency to be decoupled from the system. Further, it also allows the processes of getting and sending messages to proceed asynchronously. The send Method Request is guarded while a message is being sent.

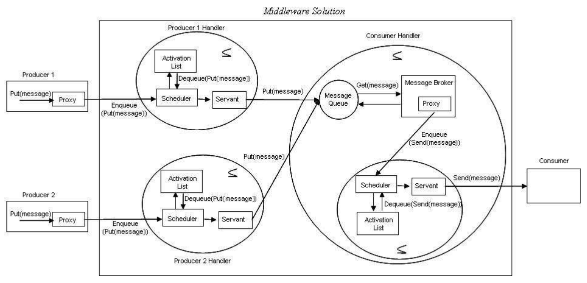

Figure 2. AO-based system

3.2 Queuing models

We assume that the arrival process at the producers is Poisson with rates λ_1 and λ_2. The times taken to put and get messages from the Message Queue remotely, over the network, are assumed to be exponential with parameter μ. The put and get times are assumed to be identically distributed for remote access, since these are governed by the network conditions, which are expected to be similar for both the producers and the consumer. Further, the internal times taken to put and get messages are assumed to be exponential with parameter τ. Since the internal access time is expected to be much lower than the remote access time, τ is at least an order of magnitude higher than μ.

Figure 3 shows the queuing model of the mutex-based system. The producers store the incoming messages in the producer-side buffers PS_1 and PS_2 until they gain access to the Consumer Handler's Message Queue, labeled MQ. The time taken by a producer to put a message on the queue and by the Message Broker to send a message to the consumer application is exponential with rate μ. A producer will not gain access to the queue if its buffer is empty or if the queue is full. Similarly, the Message Broker will not gain access to the Message Queue if the queue is empty.

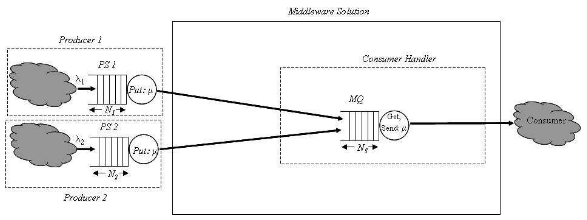

Figure 3. Queuing model: Mutex-based system

Figure 4 shows the queuing model of the AO-based system. The producer-side Activation Lists are modeled as buffers labeled $PHAL_1$ and $PHAL_2$ with capacities N_3 and N_4 respectively. A producer can continue to invoke the put method until its Activation List has spare capacity to enqueue a request. The time taken by a producer to put a message on its Activation List is exponential with rate μ. The time taken to enqueue a message on the queue internally by the producer-side servant is exponential with parameter τ. The servant can put messages on the Message Queue as long as it is not full. Also, it will not gain access to the queue if its corresponding Activation List is empty.

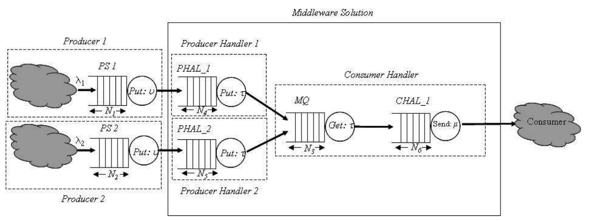

Figure 4. Queuing model: AO-based system

The consumer-side Activation List is also modeled as a buffer labeled $CHAL_1$ with capacity N_5. The time taken by the Message Broker to dequeue a message from the Message Queue is exponential with parameter τ. The rate at which the servant sends messages to the consumer is μ. The Message Broker will not gain access to an empty queue.

3.3 CSIM implementation

The implementation of queuing models using a general purpose simulation language/package such as CSIM [11] is fairly common practice. However, the implementation of the constraint of mutually exclusive access to the Message Queue in the producer/consumer systems required careful consideration and is described here. To allow synchronized access, we keep track of the threads which are "enabled" or whose constraints are satisfied and hence can gain access to the queue, in a single process that runs continuously for the entire duration of the simulation. An entity is considered to be enabled to gain access to the queue if its synchronization or guard constraints are satisfied. For example, in the mutex-based system, the producer is allowed access to the queue if there is at least one message in its buffer and if the queue is not full. This monitoring process then chooses one of the enabled entities according to a uniform distribution. It then provides the chosen entity with a semaphore, a structure called an "event" in CSIM, for the total time the entity needs access to the queue. Once the entity has completed its action on the queue, it releases the event back to the monitoring process, which then repeats the steps.

3.4 Performance metrics

In this section we define the metrics to gauge system performance. We also discuss their relevance from the user's and the provider's perspectives.

1. **Throughput:** This is the average rate at which messages are sent to the consumer application.

2. **Loss probability:** This is the average probability that an incoming message will be discarded on the producer side, due to a lack of buffer space.

3. **Response time:** This is the average time taken for a message to be received at the consumer application from the point it is created by a producer.

4. **Queue length:** This is the average queue length of the various queues in the system, namely, the producer-side queues and the Message Queue.

A service provider typically needs to balance competing concerns that consist of offering superior service performance while keeping the service cost acceptable. In a producer/consumer system, service performance will be deemed superior if the consumer application can receive messages at the same rate at which they are produced by the producers. The loss probability of the messages must thus be negligible. Further, these messages must be delivered with an acceptable response time. Thus, the first three metrics are relevant to a user's perception of performance.

To ensure acceptable service performance while maintaining reasonable costs, it is then the responsibility of the service provider to provision adequate resources. For a producer/consumer system, the storage resources consist of the various buffers used to hold messages. Thus, metric #4 can provide valuable guidance to a service provider in deciding the appropriate levels of resource provisioning.

4 Illustrations

In this section, we illustrate the potential of the methodology to guide configuration and provisioning decisions using the case study of a stock broker system [2]. The system has two producers, one each for creating NYSE and NASDAQ feeds, which we designate as producers #1 and #2 respectively. A remote data mining consumer application receives these feeds and provides stock data to the stock brokers. Since the stock brokers base important trading decisions on this data, it is extremely necessary that these feeds be received in a timely manner. Thus, the response time is a vital performance metric for the stock broker system.

For the sake of illustration, we use the nominal parameter values reported in Table 1. When using the methodology at design time, these values can be obtained for a specific hardware and operating environment either by conducting measurements on similar systems or by consultation with the experts. The performance metrics for the mutex-based and AO-based systems for the nominal values are reported in Table 2. The table indicates that both the systems have identical throughput, and is the same as the total rate at which messages are produced. The response time of the AO-based system, however, is lower than the mutex-based system. This is consistent with the average queue lengths, which are higher in the mutex-based system than the AO-based system. Thus, if the response time of the mutex-based system is unacceptable then the provider may have to disfavor the mutex-based system despite its simplicity and instead use the AO-based implementation.

Table 1. Nominal parameter values

Parameter	Value
Arrival rates (λ_1, λ_2)	15.0/sec.
Service rate (μ)	120.0/sec.
Producer-side buffers (N_1, N_2)	10
Message Queue (N_3)	100
Producer Hdlr. Activation Lists (N_4, N_5)	1
Consumer Hdlr. Activation List (N_6)	1
Internal access rate (τ)	1000.0/sec.

It is important to note that the performance of the AO-based system is better than the mutex-based system, even when the sizes of Activation Lists in the AOs are 1. When the Activation List sizes are 1, the producer blocks and cannot place any message on the Activation List until the producer-side servant gains access and puts the message it already has on the Message Queue. This shows that the AO-based implementation is effective in shielding the system from the impact of the network latency even with minimum possible Activation List sizes. This effectiveness may increase as the sizes of the Activation List increase.

Table 2. Performance of Mutex and AO systems

Metric	Mutex system	AO system
Throughput	30.00/sec.	30.00/sec.
Loss probability	0.00	0.00
Response time	0.0326	0.0227
Producer-side queue	0.261	0.146
Message queue size	0.456	0.052

Next we illustrate the utility of the methodology to enable sensitivity analysis, which is particularly valuable at design time, since at this stage the parameter values are not known with certainty. It is then crucial to determine the parameter ranges over which system performance is acceptable for a given set of configuration options. As an example, in the stock broker system frequent feeds are desirable to improve the accuracy of the data provided to the stock brokers. However, each feed must be delivered in a reasonable time. For given configuration options, the maximum rate of data feeds that can be sustained while providing an acceptable response time must then be determined. For this purpose, we vary the message arrival rate of the producers from 10.0/sec to 30.0/sec in steps of 5.0/sec. The performance metrics for both the systems as a function of the arrival rate are in Figure 5. The top left plot in the figure shows that the throughput of both the systems is identical over most of the range, except when the arrival rate is very close to 30.0/sec., at which the throughput of the mutex-based system dips slightly. As expected, the queue lengths and the response time increase as the arrival rate increases, however, the increase is more pronounced for the mutex-based system than for the AO-based system. When the arrival rate exceeds 25.0/sec. the queue lengths and the response time of the mutex-based system increase sharply, while the increase is still gradual for the AO-based system. Thus, if it is expected that the feed rates will exceed 25.0/sec., the performance of the mutex-based system may be unacceptable, mandating a switch to the AO-based system.

The above examples illustrate how the performance analysis methodology could be used to select an appropriate pattern and its configuration options to achieve acceptable system performance.

5 Related research

Performance and dependability analysis of some middleware services and patterns has been addressed by a few researchers. Aldred *et al.* [1] developed Colored Petri Net (CPN) models for different types of coupling between the application components and with the underlying middleware. They also defined the composition rules for combining the CPN models if multiple types of coupling are used simultaneously in an application. A dominant aspect of these works is related to application-specific performance modeling. In contrast, we are concerned with determining how the underlying middleware that is composed for the systems they host will perform. Kahkipuro [4] proposed a multi-layer performance modeling framework based on UML and queuing networks for CORBA-based systems. The methodology, however, is for generic CORBA-based client/server systems rather than for systems built using design patterns. The research reported in this paper is concerned with performance analysis of a specific design pattern used in the development of producer/consumer systems. The work closest to our work is in [8], where a performance model of the CORBA event service is developed. Our prior research has focused on performance analysis methodologies for other middleware patterns including the Reactor [3] and the Proactor patterns [7].

6 Conclusions and future research

In this paper we presented a model-based performance analysis methodology for an AO-based system. It comprised of a queuing model which captured the internal architecture of an AO-based system. A CSIM implementation of the model was used to demonstrate the utility of the methodology in guiding provisioning and configuration decisions on an example stock broker system.

Our future research consists of developing an analytical/numerical approach for the performance analysis of the AO pattern. Developing a strategy to compose the performance models of patterns mirroring their composition on the middleware stack is also a topic of future research.

Acknowledgments

This research was supported by the following grants from NSF: Univ. of Connecticut (CNS-0406376, CNS-SMA-0509271) and Vanderbilt Univ. (CNS-SMA-0509296).

References

[1] L. Aldred, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. “On the notion of coupling in

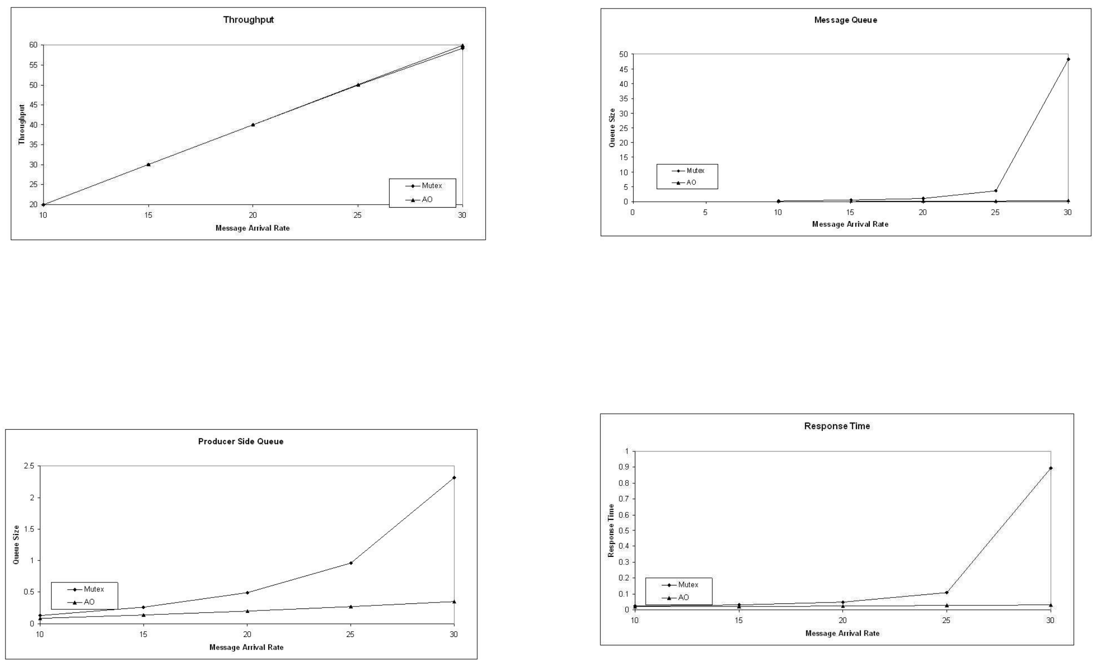

Figure 5. Sensitivity of performance measures to message arrival rates (λ_1, λ_2)

communication middleware". In *Proc. of Intl. Symposium on Distributed Objects and Applications (DOA)*, pages 1015–1033, Agia Napa, Cyprus, 2005.

[2] G. Banavar, T. Chandra, R. Strom, and D. Sturman. "A case for message oriented middleware". In *Proc. of the 13th Intl Symposium on Distributed Computing*, pages 1–18, London, UK, 1999. Springer-Verlag.

[3] S. Gokhale, A. Gokhale, and J. Gray. "Performance analysis of a middleware demultiplexing pattern". In *Proc. of Hawaii Intl. Conference on System Sciences (HICSS)*, January 2007.

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

[5] R. Greg Lavender and Douglas C. Schmidt. *Pattern languages of program design 2*, chapter Active object: an object behavioral pattern for concurrent programming, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1996.

[6] ORACLE. "Bringing SOA value patterns to life". White Paper, June 2006.

[7] U. Praphamontripong, S. Gokhale, A. Gokhale, and J. Gray. "Performance analysis of an asynchronous Web server". In *Proc. of Intl. Conference on Computer Science and Applications*, pages 22–25, 2006.

[8] 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.

[9] R. E. Schantz and D. C. Schmidt. "Middleware for distributed systems: Evolving the common structure for network-centric Applications". In John Marciniak and George Telecki, editors, *Encyclopedia of Software Engineering*, pages 801–813. Wiley & Sons, 2002.

[10] 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.

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