Performance of Middleware Based Architectures: A Quantitative Approach[1]

M. Harkema [a,c], B.M.M. Gijsen[b] and R.D. van der Mei[c,d]

[a] University of Twente, Department of Computer Science, Enschede, The Netherlands
[b] TNO Telecom, Center of Excellence Quality of Service, Delft, The Netherlands
[c] CWI, Advanced Communication Networks, Amsterdam, The Netherlands
[d] Vrije Universiteit, Faculty of Sciences, Amsterdam, The Netherlands

Keywords: middleware, performance, quantitative models, simulations, lab testing, thread pools, thread scheduling

Most of today's E-business applications on the Internet are built upon middleware-based architectures. For service providers offering these applications performance is essential: less-than-acceptable performance levels may lead to customer churn, and thus loss of revenue, and as such directly affect the company's competitive edge. This raises the critical need for service providers to be able to predict and control performance. In this paper we demonstrate the usefulness of a quantitative modeling approach to analyze and predict the performance of middleware-based applications. To this end, we develop a quantitative performance model of middleware architectures based on CORBA, the de-facto standard for object middleware. A particular feature of the model is that it explicitly takes into account priority mechanisms that handle the access to the processors among the different threads. To validate the model we have compared performance predictions from simulation runs with results from lab experiments for a variety of parameter settings. The results show that (1) the inclusion of priority mechanisms in the model leads to a significant improvement of the accuracy of the performance predictions based on the model, and (2) a quantitative modeling approach to assess and predict the performance of middleware-based applications is very promising.

1. INTRODUCTION

Many of today's E-business applications on the Internet run in a heterogeneous environment of networks, hardware and software components. In the competitive market of E-businesses a critical success factor service providers is performance. Performance problems can directly lead to customer churn, and thus loss of revenue. Typical examples of E-business applications are online airline ticket reservation, online banking and online purchasing of consumer products. For this type of applications, the most relevant performance aspects are service availability, payment transaction security and performance. This paper is focused on performance, particularly in terms of response times.

To assess the performance of their E-business applications, companies usually perform a variety of activities: (1) performance lab testing, (2) performance monitoring, and (3) performance tuning. Lab testing typically involves the performing load and stress testing in a lab environment. Although lab-testing efforts are undoubtedly useful, there are two major disadvantages. First, building a production-like lab environment may be very costly, and second, performing load and stress tests and interpreting the results are usually very time consuming, and hence highly expensive. Performance monitoring is usually performed to keep track of high-level performance metrics such as service availability and end-to-end response times, but also to keep track the consumption of low-level system resources, such as CPU utilization and network bandwidth consumption. Results from lab testing and performance monitoring provide input for tuning the performance of an application. A common drawback of the aforementioned performance assessment activities is that their ability to predict the performance under projected growth of the workload in order to timely anticipate on performance degradation (e.g., by planning system upgrades or architectural modifications) is limited. This raises the need to complement the activities with methods specifically developed for performance prediction [7]. To this end, various modeling and analysis techniques have been developed over the past few decades (e.g., see [4, 6, 9] and references therein).

In this paper, we develop a quantitative model for the performance of a CORBA-based middleware implementation. The model encompasses the combined impact of a variety of factors, such as the processor speed, the rate at which requests arrive at the server, the marshalling and un-marshalling of requests, de-multiplexing the request to the proper application object, and the priority mechanisms implemented to schedule the different

[1] This work was partly carried out within the EQUANET project, an "ICT-doorbraakproject" which is supported by the Dutch Ministry of Economic Affairs via its agency SenterNovem. Information on the EQUANET project is available from http://equanet.cs.utwente.nl/.

types of threads at the processor. As such, the model is an extension of the model presented in [1], where we assumed that all the active threads share the underlying processor speed in a processor-sharing (PS) fashion, not taking into account some kind of priority mechanism. We validate the model by comparing results from lab experiments with simulation results for a number of workload scenarios. The results demonstrate that the performance predictions based on the model match well with the results from the lab experiments. Moreover, comparing the results with those in the model without priorities [1] the results show that the inclusion of the priority mechanism to scheduling threads leads to significant enhancements of the accuracy of the model.

The remainder of this paper is organized as follows. Section 2 describes the performance model. Section 3 compares performance experiment results with simulation results. Section 4 presents our conclusions.

2. PERFORMANCE MODEL

In this section we develop a quantitative performance model for remote method invocations based on CORBA. Compared to our previous paper on this performance model [1], we have added features of the operating system scheduler where we previously assumed PS scheduling. Large parts of section 2.1 and 2.2 have been preserved to keep this paper self-containing.

2.1 Description of request handling

In this section we describe how the processing steps in handling requests (see for example [1, 10] for more details) are handled by the operating system and middleware layer. The focus is on the request handling at the server side, i.e. the handling of method invocation requests, which is essential for most CORBA applications.

In the discussion below the middleware is configured to use the *thread pool* ORB threading model. In this threading model there are two types of threads: *receiver threads* that receive incoming requests from the network and *dispatcher threads* that dispatch these requests onto the target object implementation. The ORB allocates a pool of dispatcher threads during startup. This *thread pool* has a fixed size.

Consider a server that receives and handles requests that come in over one of the N_{con} connections. To discuss the functional behavior of the request handling mechanism, let us consider a tagged method invocation request T and follow its route along the successive processing steps, illustrated by Figure 1.

To start, upon entering the system T is received by a *receiver thread*, which is used to perform several processing steps. Specifically, to read the header of the request, to read the body of the message, to search in the active object map for the object key, to locate the so-called Portable Object Adapter (POA) belonging to the invoked object and to send the request to the dispatcher thread pool. Newly incoming requests that arrive at a busy receiver thread are queued and served in the order of arrival. After finishing the receiver-thread processing, the receiver thread is released and the method invocation request T is forwarded to a pool of *dispatcher threads* that handle access to the POAs and object implementations. The receiver thread en-queues request T in the request queue of the dispatcher thread pool, even if there are available dispatcher threads and the queue is empty. After en-queuing the request T, idle dispatcher threads are signaled that a new request is available in the queue. After these so-called 1st phase receiver thread processing steps the receiver thread is not ready to process a new request, until the 2nd phase has been completed. The 2nd phase consists of cleaning allocated data-structures and preparing to process for the next request. Once a dispatcher thread is available (after the receiver thread signaled the new request T in the queue) request T is sent to the proper POA, which contains a reference to the object that will handle the request. The POA may be configured to use a single-threaded or multi-threaded policy. With the single-threaded policy only one request can be processed by that POA. Other dispatcher threads that also want to process request on that POA will block until the POA becomes available again. Each ORB has a Root-POA with a standard collection of policies, for instance the Root-POA has the multi-threaded threading policy. Subsequently, the POA sends the request to the skeleton, which un-marshals the request and sends the request to the *server object implementation* (also referred to as servants) that will handle the request. Finally, the reply of the object execution is *marshaled* and sent back to the client and the dispatcher thread is released. Similar to the operation of a receiver thread, the dispatcher thread processing is completed by a second phase part where data-structures are cleaned and preparations are made to process the next request.

2.2 Model

Method invocation requests arrive at the server over one of the N_{con} connections. Method invocation requests for object j at POA i arrive at the server according to a Poisson process with rate $\lambda_{i,j,k}$ requests per time unit. To describe the dynamics of the model, we consider a tagged customer $T=T_{i,j,k}$ and follow its route along the different processing steps. Each receiver thread serves incoming requests in the order of arrival, and requests finding the receiver thread busy have to wait in an infinite-size buffer. Processing the request by the receiver thread takes a mean amount b_{rec} of CPU processing (same for all i); this processing time is assumed to be deterministic and includes both 1st and 2nd phase CPU time. After being processed the received thread, the receiver thread is released and T is forwarded to the dispatcher thread pool, consisting of N_{disp} dispatcher threads. If T finds all N_{disp} dispatcher threads occupied it is placed in a infinite-size buffer that is handled on a first-come-first-served basis. When a dispatcher thread is available, T is sent to the proper POA (namely, POA i, which is predetermined upon arrival). The amount of service time required at the POA is the deterministic CPU processing time b_{poa} (including 1st and 2nd phase CPU time). If i=1, then T is forwarded to the Root-POA and taken into service immediately; otherwise, T is forwarded to POA i and handled on a FIFO basis. Subsequently, POA i forwards T to servant j. In practice, POAs and servants may be single- or multi-threaded with any number of threads. In our performance model the threading level of POA i

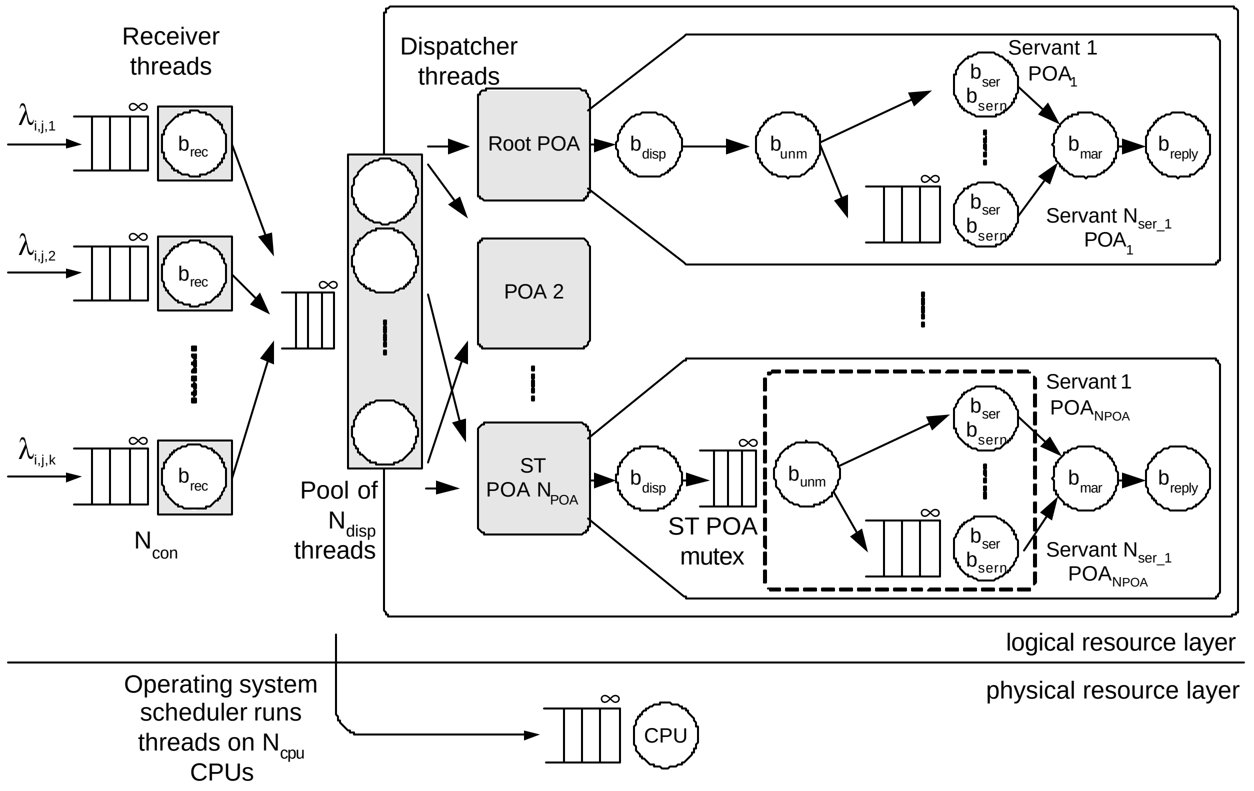

Figure 1. Performance model for the server-side request handling.

and servant j at POA i is represented by T_{poa_i} and $T_{ser\_i,j}$, respectively. The amount of processing time needed by T consists of a mean amount of CPU processing time $b_{ser\_i,j}$ and a mean non-CPU processing time of $b_{sern\_i,j}$ for object j at POA i (deterministically distributed). Non-CPU processing time represents idle times, database access times, memory access times, disk I/O, etcetera. The reply of the method invocation will be marshaled after the object has processed the request. The processing time needed for the marshaling consists of a deterministic amount of CPU processing time $b_{mar\_i,j}$ for the marshaling after object j belonging to POA i. The precise amount of deterministic CPU processing time for marshaling depends on the amount and type of data that is to be marshaled. As soon as the reply is sent, the POA and dispatcher thread are released. In other words, the dispatcher thread is possessed by the request during the POA, servant and marshaling steps.

The processing steps performed by the receiver and dispatcher threads share the hardware resources. In our first version [1] of the performance model we assumed that the threads shared the CPU in a PS fashion. That is, if at some point in time there are in total N receiver and dispatcher threads active, then each of them receives a faction of 1/N of the available processor capacity (on a single CPU machine).

The second version of the performance model more accurately captures the thread scheduling behavior of the Linux operating system. Linux schedules thread execution using a time sharing scheduler with variable quantum of 10 – 110 milliseconds. The scheduling order of threads depends on the time-slice threads have left. When there are no runnable (non-blocking) threads left with a time-slice larger than zero, the time-slices of all threads are recalculated. Threads keep half of their remaining time-slice from the last scheduler round and some constant amount of time-slice is added. By allowing threads to keep half of their remaining time-slice, Linux favors threads that don't spend a lot of time on the CPU (i.e. sleeping / blocking threads). In other words, Linux favors I/O bound threads over CPU bound threads. Thread context switches occur when a thread has no time-slice left, when a thread blocks for some resource, or when a thread with a larger time-slice un-blocks (e.g. a dispatcher thread un-blocking because a receiver thread en-queued a new request in the thread-pool queue preempts the receiver thread if its time-slice is larger). Every 10 milliseconds the kernel decrements the time-slice of the running thread.

More detailed information on the Linux scheduler is available from [8] and the Linux kernel source code.

3. MODEL VALIDATION

In this section we describe our test lab setup and present some performance experiment results with accompanying simulation results, using both our previous [1] and updated performance simulation implementations.

3.1 Test lab setup and measurements

Our test lab consists of 2 machines interconnected using local network. The server machine is a Pentium III 550 MHz with 256

Threads	*Measured receiver queuing time (ms)*	*Ruby*	*Extend*	*Measured receiver completion time (ms)*	*Ruby*	*Extend*	*Measured dispatcher queuing time (ms)*	*Ruby*	*Extend*	*Measured dispatcher completion time (ms)*	*Ruby*	*Extend*
1	0.53	0.05	0.00	0.94	0.97	1.69	6.71	7.39	7.25	6.30	6.25	6.36
2	0.59	0.12	0.01	0.98	1.04	2.10	5.45	5.83	5.23	8.86	8.44	10.32
4	1.55	1.52	0.01	1.61	2.27	2.52	3.57	3.80	1.68	10.14	10.07	14.62
6	3.59	4.36	0.02	3.55	4.90	2.69	1.86	1.21	0.66	8.65	7.91	16.45
8	5.39	5.22	0.03	5.55	6.00	2.74	0.74	0.47	0.37	6.84	6.79	16.95
10	6.01	6.15	0.03	6.04	6.41	2.73	0.40	0.13	0.10	6.36	5.97	17.16
15	6.40	6.36	0.03	6.59	6.61	2.71	0.09	0.00	0.00	5.84	5.64	17.08
20	6.25	6.51	0.02	6.69	6.58	2.62	0.04	0.02	0.00	5.75	5.69	16.52
25	6.53	6.49	0.02	6.67	6.61	2.63	0.04	0.00	0.00	5.73	5.64	16.43

MB RAM. The client machine is a Pentium IV 1.7 GHz with 512 MB RAM. Both machines run the Linux v2.4 operating system and the Java 2 standard edition v1.4.1. For this experiment we disabled priority scheduling of processes on the Linux machines and used high-resolution timers to generate accurate arrival processes. The CORBA implementation we use is ORBacus 4.1.1 by IONA Technologies [3]. In the experimental setup one target object, managed by the Root-POA, is instantiated in each scenario. The client machine runs a synthetic workload generator [1] that produces workload for the CORBA implementation running on the server machine. To obtain measurement data of the CORBA server, we used our Java Performance Monitoring Toolkit (JPMT) [2]. Amongst other measures, we monitored the following performance data during the experiments:

- Queuing times before the receiver thread and the dispatcher thread-pool.
- CPU times of the receiver and dispatcher threads.
- Completion times of the receiver and dispatcher threads.
- CPU utilization.

3.2 Performance results

We configure the CPU service demand of the target object to be 5 milliseconds with a deterministic distribution. The server ORB dispatcher thread pool has 10 threads. The client ORB will generate a workload of 15000 requests using a Poisson process, via one connection. The arrival-rate for the Poisson process is 1 request per 10 milliseconds.

Table 1 contains a summary of the performance measurement results, together with simulation results from both our previous and updated simulators (discussed in section 3.3). The presented values are all averaged over the 15000 requests. The measured receiver queuing time represents the time that requests are queued before the receiver thread (somewhere between client and server-side receiver thread, e.g. in socket buffers). The measured dispatcher queuing time represents the time that requests are queued at the dispatcher thread-pool. The measured completion time of the receiver and dispatcher threads equal the (average) time between the arrival at the thread and departure of a request from the thread, after processing the request. The CPU consumption of the receiver thread is 0.925 milliseconds per request. A dispatcher thread consumes 5.6 milliseconds per request, including the 5 milliseconds of the method executed by the target object.

The measured queuing and completions times of the receiver and dispatcher threads are illustrated by Figures 2 and 3, together with simulated values from the updated performance model.

The performance experiment results show that different numbers of dispatcher threads cause different queuing behavior, while not really influencing overall request response times (around 14 milliseconds).

3.3 Simulation results

We first implemented the performance model described in section 2 in the Extend [5] simulation tool. We modeled thread scheduling behavior using a PS node. This implementation yielded encouraging performance predictions [1], but overestimated the completion times of the dispatcher threads and underestimated receiver thread completion times and queuing times for the receiver and dispatcher threads. As shown in Table 1, the dispatcher thread completion times in the Extend simulation are much higher than measured during the experiments. These higher completions times are caused by the PS scheduling, which effectively schedules all runnable threads with an infinitely small time quantum. The overall response times were over estimated by the Extend simulation, mostly because of overestimation of the dispatcher thread completion times.

Our new simulation model is implemented using the Ruby programming language. Besides the middleware features described in section 2, this simulation model also implements the operating system scheduler features described in section 2. As shown in Table 1 and Figures 2 and 3, the predictions of both queuing times of receiver and dispatcher threads and completion times of the receiver and dispatcher thread have become more accurate.

4. CONCLUSIONS

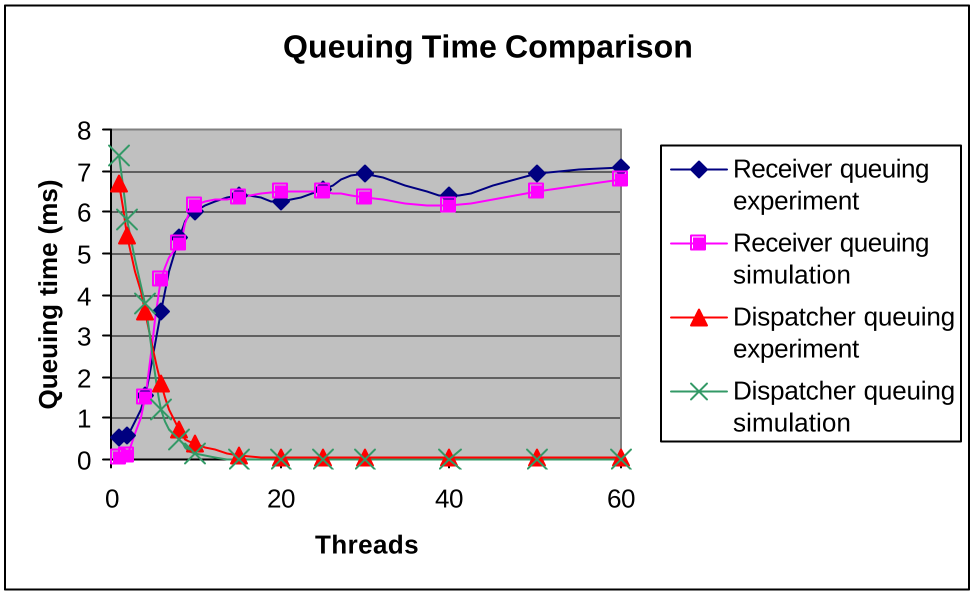

Figure 2. Receiver and dispatcher thread queuing time comparison.

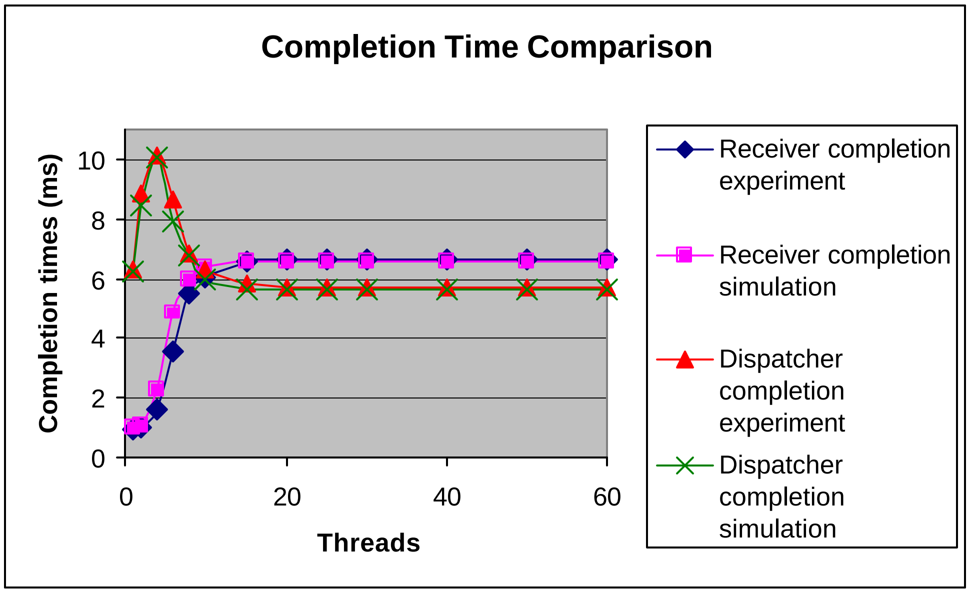

Figure 3. Receiver and dispatcher thread completion time comparison.

In this paper we refined our quantitative performance model for middleware architectures based on CORBA, first presented in [1]. A particular feature of the updated model is that it explicitly takes into account priority mechanisms that handle the access to the processors among the different threads. Validation results of the updated model show accurate results of queuing times and thread completion times, whereas in our previous model these were often under- or over-estimated.

5. REFERENCES

[1] M. Harkema, B.M.M. Gijsen, R.D. van der Mei and Y. Hoekstra. *Middleware Performance: A Quantitative Modeling Approach*. To appear in Proc. of the international Symposium of Performance Evaluation of Computer and Telecommunication Systems (SPECTS), San Jose, July 2004.

[2] M. Harkema, D. Quartel, B.M.M. Gijsen, R.D. van der Mei, *Performance Monitoring of Java Applications*, Proc. of the 3rd Workshop on Software and Performance (WOSP), 2002.

[3] IONA Technologies, Object Oriented Concepts Inc., *ORBacus 4 for Java*, 2000.

[4] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons, 1991.

[5] D. Krahl, Imagine That Inc., *The Extend Simulation Environment*, Proceedings of the 2000 Winter Simulation Conference, Orlando, FL, USA, 2000.

[6] E.D. Lazowska, J. Zahorjan, G.S. Graham, K. Sevcik, *Quantitative System Performance*, Prentice-Hall Inc., 1984.

[7] R.D. van der Mei, B.M.M. Gijsen and J.L. van den Berg, *End-to-end Quality of Service modeling of distributed applications: the need for a multidisciplinary approach*, CMG Journal on Computer Management 109, 51-55, 2003.

[8] D.P. Bovet, M. Cesati, Understanding the Linux Kernel, 2nd edition, O'Reilly Media, Inc., 2002.

[9] R. Sahner, K.S. Trivedi, A. Puliafito, *Performance and Reliability Analysis of Computer Systems*, Kluwer Academic Publishers, 1996.

[10] S. Vinoski, *CORBA: Integrating diverse applications within distributed heterogeneous environments*, IEEE Communications Magazine, February, 1997.