Middleware Performance: A Quantitative Modeling Approach

M. Harkema[1] B.M.M. Gijsen[2] R.D. van der Mei[2,3], Y.Hoekstra[3]

[1] Department of Computer Science
University of Twente
The Netherlands
m.harkema@cs.utwente.nl

[2] Expertise Group QoS Control
TNO Telecom
The Netherlands
b.m.m.gijsen@telecom.tno.nl

[3] Faculty of Exact Sciences
Vrije Universiteit Amsterdam
The Netherlands
mei@cs.vu.nl

ABSTRACT

Middleware is software that manages interactions between applications distributed across a heterogeneous computing environment. Middleware technology is often used to implement E-business applications on the Internet. Performance problems in these distributed E-business applications can lead to customer churn, and thus loss of revenue. This raises the critical need for service providers to be able to predict and control performance. Motivated by this, in this paper we develop a quantitative performance model of CORBA-based middleware. CORBA is the de-facto standard for object middleware, providing RPC-like interactions between objects. We have implemented the performance model in a simulation tool. 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 the model leads to accurate performance predictions, and moreover demonstrates that a quantitative modeling approach to assess and predict the performance of middleware-based applications is very promising.

Keywords

Middleware, Java, performance measurement, performance modeling, simulation

1. INTRODUCTION

The dramatic growth of the Internet, the ongoing developments in the hardware and software industry, and the recent advances in networking technology have boosted the emergence of Information and Communication Technology (ICT). ICT systems enable applications to be divided in components that can be executed on geographically distributed information systems. These applications are commonly referred to as distributed applications. Distributed computing provides the fundamental technology for the realization of enterprise-wide and even global information systems. This development has led to the emergence of a wide variety of E-business applications that have been brought to the market. In this competitive market of E-businesses a critical success factor for E-business applications is the Quality of Service (QoS). QoS 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 QoS aspects are service availability, payment transaction security and performance. This paper is focused on performance, particularly in terms of response times.

Distributed applications typically run in a heterogeneous environment of networks, hardware and software components. Middleware architectures have been developed to shield developers of distributed application from the problem interoperability problems. Middleware is software that resides between the application and the operating system. As such, middleware performance is an important part of the end-to-end performance of distributed applications. The performance of distributed middleware-based applications depends on many factors, including network performance, the performance of the application code and middleware, and the performance of the hardware on which the application and middleware is being executed.

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 [12]. To this end, various modeling and analysis techniques have been developed over the past few decades (e.g., see [10, 8, 14] and references therein).

The de facto middleware standard is the Common Object Request Broker Architecture (CORBA) developed by the Object Management Group (OMG), an international consortium of companies and institutions. 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 processing times of the middleware (including for instance the overhead induced by de-multiplexing the request to the proper application object, and the (un-)marshaling costs) and the processing times of the application. We validate the model by comparing results from lab experiments with simulation results for a number of workload scenarios. The results demonstrate that the accuracy of the performance predictions based on the model match well with the results from the lab experiments.

While we focus on one CORBA middleware implementation, the model is still a useful basis for modeling other CORBA implementations, since they are often similar in design and implementation. To a lesser extent, the paper could be useful for insights on how to model other middleware architectures, such as DCOM [3] and Enterprise Java Beans (EJB) [11].

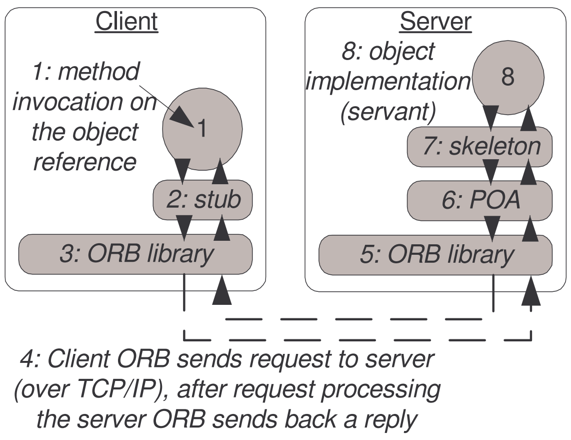
In the literature, the development of quantitative performance models for middleware servers has received little attention. As an exception, Sheikh et. Al. [15] develop a layered queuing model and a simulation model for a large-scale distributed network management application based on CORBA middleware. The layered queuing model was validated with respect to the simulation model. In [1] three architectures for interaction between client and servers in a CORBA system are described and performance of them is compared under several workload conditions.

We emphasize that the focus of this paper is on the development of a quantitative performance model that encompasses the main factors that impact the performance. The model proposed in this paper is not in the class of 'simple' (layered) queuing models, for which straightforward solution techniques exist. In particular, model features such as a finite thread pool and deterministic service times (as will be described in the next section) prohibit straightforward application of analytic techniques. Due to this observation *and* our focus on performance modeling instead of model analysis, we resort to analyzing the presented performance model by means of simulation in this paper. The ultimate goal of the work presented here, is to apply the developed performance model for predicting the performance of specific middleware based applications. As such, the work presented in this paper is a first step towards this ultimate goal.

The remainder of this paper is organized as follows. Section 2 describes the performance model. Section 3 describes our test lab setup and performance experiments. Section 4 briefly describes the implementation of the performance model in a simulation tool and the performance results we have obtained from simulation runs. Section 5 discusses the validation of the performance model by comparing the performance results from lab experiments with simulation results. Section 6 presents our conclusions and addresses several topics for further research.

2. PERFORMANCE MODEL DESCRIPTION

In this section we develop a quantitative performance model for remote method invocations based on CORBA, the de-facto object-middleware standard. To this end, in section 2.1 we describe the sequence of steps involved in invoking a method on a CORBA object. In 2.2 we focus on the server-side handling of CORBA method invocation requests on application objects, and discuss the threading models involved. Based on this, in 2.3 we propose a quantitative performance model for CORBA server-side request handling. Section 2.4 contains a discussion of the pros and cons of the modeling assumptions.



Client
Server
1: method invocation on the object reference
1
2: stub
3: ORB library
8: object implementation (servant)
8
7: skeleton
6: POA
5: ORB library
4: Client ORB sends request to server (over TCP/IP), after request processing the server ORB sends back a reply

Figure 1. Anatomy of a CORBA method invocation

2.1 Anatomy of a CORBA method invocation

To invoke a method on a remote CORBA object, the following sequence of steps is taken, as illustrated by Figure 1 (illustrating a two-way request-and-reply method invocation).

1. **Remote method invocation**. The client obtains the object reference of the remote target object and performs a method invocation on it as if the object were a local (e.g. Java) object.
2. **Stub processing and marshaling**. What really happens is that the client invokes the method on the *stub*, which is the local proxy of the remote target object. A reference to the proper stub is obtained from the object reference of the target object. The stub constructs a CORBA request object and translates the method invocation parameters, which are expressed using programming language, operating system, and architecture specific data types, to a common data representation (CDR). This translation process is called *marshaling*. The marshaled data is added to the request object. Subsequently, the request object is forwarded to the client-side object request broker (ORB) library.
3. **Client-side ORB processing**. The client-side ORB library uses a TCP/IP connection to communicate with the server-side ORB library. The address of the server-side ORB is obtained from the target object's *object reference*. The object reference was created by a portable object adapter (POA) in the server-side ORB. Each object is managed by exactly one POA. A POA implements the adapter design pattern [2] to adapt the programming language specific object interfaces to CORBA interfaces, making the target object implementation accessible from the ORB. The POA has a map of active objects. This map associates object identifiers with object implementations. Object implementations are called *servants*. The object reference contains server information, such as hostname and port

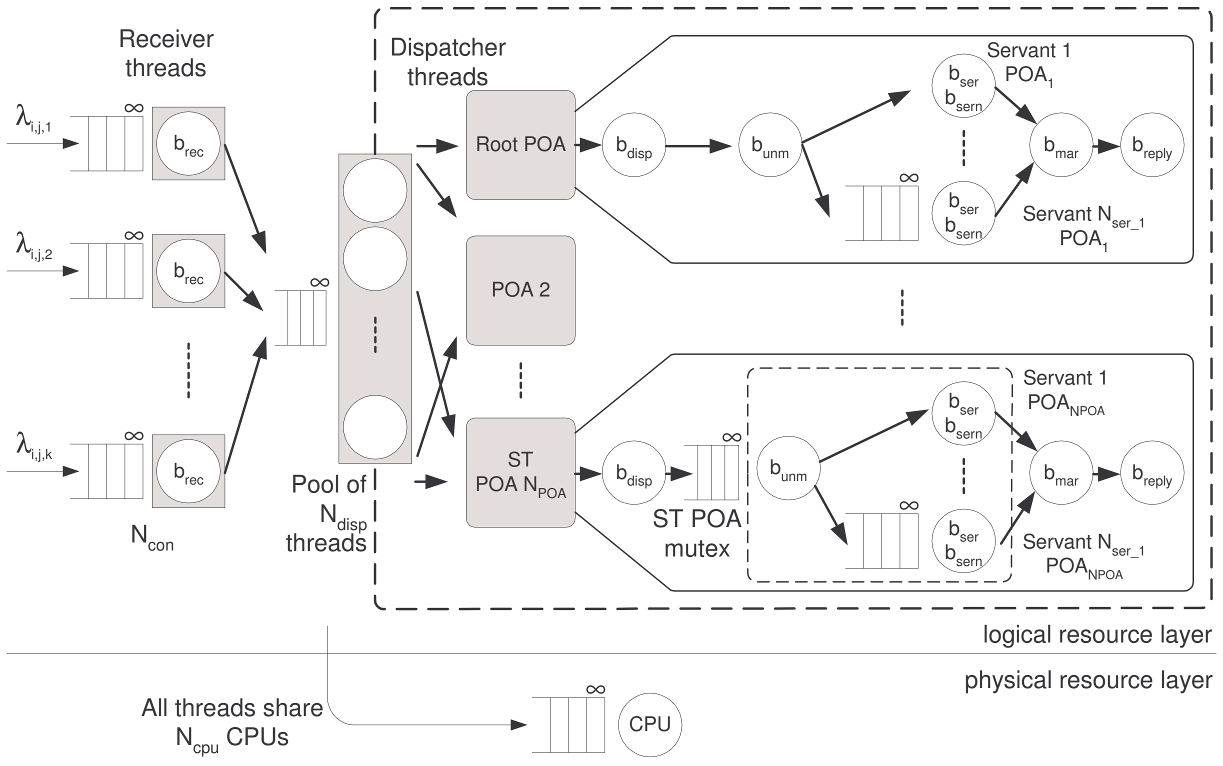


Figure 2. Performance model for the server-side request handling. The receiver threads and dispatcher threads share a CPU resource.

number, the name of the POA, and the object identifier of the target object.

4. **ORB communication**. The client-side ORB sends the request object to the server-side ORB.
5. **Server-side ORB processing**. The server-ORB obtains the target object's POA and object identifiers from the request object, and forwards the request to the POA managing the target object.
6. **Object adaptor processing**. The POA looks up the servant in its *active object map* (a data-structure containing references to the active objects) using the object identifier and forwards the request to the skeleton of the target object.
7. **Un-marshaling**. The skeleton un-marshals the method invocation parameters and looks up the method implementation.
8. **Method invocation in the object implementation.** The skeleton invokes the request on the proper method implementation inside the object implementation of the target object.
9. **The road back, creating a reply and sending it to the client**. When the method invocation returns, the skeleton creates a CORBA reply object, marshals the return parameters, and inserts the return parameters in the reply object. The reply object is forwarded to the server-side ORB. Subsequently, the server-side ORB forwards the reply object to the client-side ORB. Then, the client-side ORB forwards the reply object to the stub, and finally, the stub un-marshals the return parameters and forwards those to the client.

2.2 Functional description of the server-side request handling

In this section we describe how the subsequent processing steps discussed in 2.1 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 kinds 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 2.

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.

Note that both the POAs and the objects may be defined as either single or multi-threaded, depending on the number of requests that can be handled simultaneously. The client may use synchronous (the client blocks until it receives a reply) and asynchronous requests (the client can continue its processing and poll for a reply at a later point in time). Also, the requests can be two-way (the default), meaning a reply is sent back to the client, or one-way, where no reply is sent back to the client.

2.3 Performance model of the server-side request handling

In this section we use the observations discussed in 2.1 and 2.2 to propose a quantitative performance model for CORBA server-side request handling. The request handling process is highly complex, and many factors are implementation specific. In this context, we emphasize that the goal is *not* to develop a complicated performance model that covers all relevant implementation-dependent features, but to develop a *generic* quantitative model that on the one hand covers the main factor that impact the performance of CORBA method invocation requests, but on the other hand is still simple enough to provide insight in the behavior of the CORBA request handling process, and the trade-offs involved. To this end, in some cases we have to make several simplifying assumptions. The model to be presented below should be judged from that perspective.

Model parameters

N_{con}	The number of connections to the server.
N_{poa}	The number of POAs.
N_{ser_i}	The number of servants at POA i.
T_{disp}	The number of threads available for request dispatching.
T_{poa_i}	The number of threads available for POA i.
$T_{ser\_i,j}$	The number of threads available for object j at POA i.
$\lambda_{i,j,k}$	Arrival rate for requests at connection k for object j at POA i.
b_{rec}	Mean CPU processing time needed by a receiver thread.
b_{poa}	Mean CPU processing time needed by a POA.
$b_{ser\_i,j}$	Mean CPU processing time needed by object j at POA i.
$b_{sern\_i,j}$	Mean non-CPU processing time needed by object j at POA i.
$b_{mar,\, ij}$	Mean CPU processing time needed for marshaling the reply.

For each of the indices i, j, k the following conditions hold if no other condition is mentioned: $1 \le i \le N_{poa}$, $1 \le j \le N_{ser\_i}$ and $1 \le k \le N_{con}$.

Model description

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 T_{disp} dispatcher threads. If T finds all T_{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 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 threads, the POAs and the servants *effectively share* the hardware resources. To incorporate the effect of sharing processors, we assume that all active receiver threads, POAs and servants share the processor in a processor sharing (PS) fashion. That is, if at some point in time there are in total N receiver threads, POAs and servants active at that time, then each of them receives a fraction 1/N of the available processor capacity (on a single CPU machine).

2.4 Discussion of the performance model

In section 2.3 we have proposed a model for the performance of CORBA-based method invocations. To this end, we have made a variety of simplifying assumptions. The validity and motivation for these assumptions will be addressed below. We reemphasize that the performance of CORBA-based method invocation requests is a complex interplay between the operating systems, threading models, hardware and all kinds of implementation-specific details, making our efforts to develop generic and simple-but-accurate performance models highly challenging, and that the modeling assumptions should be judged from that perspective.

2.4.1 Other threading models

First, in most Java CORBA implementations each POA manager endpoint has an *acceptor* thread, which listens for new connections. Each new connection gets a new *receiver* thread, which receives incoming requests. In some threading models the thread that receives the requests from the network also dispatches the request onto the target object implementation. In other threading models the thread that receives the request forwards the request to a separate *dispatcher* thread. The model discussed above assumes that the receiver threads handle incoming requests and forward them to a pool of dispatcher threads. This is commonly referred to as the *thread pool* policy. There are a variety of other threading policies available. For example, in the so-called *threaded* policy the receiver thread also dispatches the request. At most one thread may be active in user code. Other receiver threads that want to dispatch requests will have to wait. Alternatively, the *thread-per-connection* model is similar to the *threaded* model. The only difference is that at most one thread per connection may be active in user code. In practice the *threaded* model has an additional mutex, which serializes all threads that want to dispatch to user code. As another alternative, in the *thread-per-servant* model the requests are received by receiver threads, but dispatched to a separate thread for each servant. This dispatcher thread per servant has a request code. At most one request is active for each servant. In the *thread-per-request* model requests are received by receiver threads, but each request is dispatched to a newly created dispatcher thread, i.e. a new dispatcher thread is created for each request. Finally, the *leader/follower thread pool* model is an optimization of the basic thread pool model. Instead of adding the request to the request queue of the dispatcher thread pool, the receiver thread (the leader) becomes a dispatcher thread itself. One of the idle threads (followers) in the leader/follower thread pool becomes the new receiver thread (thus the new leader). The optimization is that a context switch from receiver to dispatcher thread is saved.

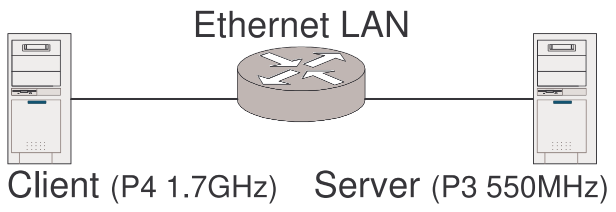


Figure 3. The setup of our test-bed. The client host invokes CORBA methods on the server host with varying request payload.

2.4.2 Modeling assumptions

In the model described above, several assumptions have been made. In this context, it is important to realize that in practice, detailed information about actual system parameters is lacking, at best limited, so that first-order assumption are usually the best one can do. The assumptions addressed below should be viewed in this context.

Arrival processes of method invocation requests. The model assumes that 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. Experience has taught that this assumption is reasonable in most cases, and also has the benefit of being characterized by a single parameter per servant. In cases where requests over a given connection arrive according to a more bursty arrival patterns, the Poisson assumption may be directly relaxed and refined, depending on the amount of information available.

Sharing of processor capacity. In the model it is assumed that the processing steps performed by the receiver threads, the POAs and the servants *effectively share* the hardware resources in a processor sharing fashion, where each active thread receives a fair share of processor capacity. This assumption is clearly a first-order approximation, but in most cases "the best we can do", given the lack of available information about implementation-specific details about the scheduling of threads and processes on the processors. Incorporating such implementation-specific details is certainly possible and is expected to improve the accuracy of the model, but will undoubtedly go to the expense of the transparency of the model.

3. PERFORMANCE EXPERIMENT RESULTS

Since performance models provide only predictions, and not observations, modeling results can become inaccurate if the model does not capture the performance behavior of the applications closely enough. Building accurate performance models of a system requires insight in the internal behavior of the system and good quality performance measurements. To obtain the insight and measurements, we conduct performance experiments in a test lab.

In this section we describe how we conduct performance experiments in a test lab and present results from a couple of scenarios.

3.1 Test lab setup

Our test lab consists of 2 machines interconnected using local network. The server machine is a Pentium III 550 MHz with 256 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 [7]. 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 that produces workload for the CORBA implementation running on the server machine. The synthetic workload generator described in section 3.2. We use our monitoring toolkit, described in section 3.3, to obtain performance measurements for the CORBA implementation running on the server machine.

3.2 Generating workload

We developed a workload generator so that we could automate performance experiments with different scenarios. The workload generator consists of a client and server application.

The server-side application offers the following scenario configuration options:

- Specification of the POA hierarchy, including POA managers and POA policies (e.g. single-threaded POAs).
- Deployment of objects on the specified POAs.
- Service demands for methods in the object implementation. Both CPU time usage and waiting time can be described. The CPU time can be used to work that is done by the object implementation. The waiting time can be used to simulate that the object implementation is waiting for an external entity, for instance a query to a remote SQL database.
- Configuration options for the ORB, for instance which threading model to use or connection reuse policies.

The client-side application executes a given workload on the server application. The workload description consists of a collection of arrival processes. An arrival process description consists of:

- The total number of requests to generate.
- The targets of the arrival process. A description of a target consists of the name of the remote object, the name of the method to invoke, and (if applicable) requests parameters (payload). If an arrival process has multiple targets, then by default the requests are equally distributed over the targets. It is possible, however, to specify routing probabilities for each target.
- The request arrival process can either a Poisson process (exponentially distributed inter-arrival time) or deterministic (e.g. exactly 1 request every 2 seconds). Requests are generated regardless of the completion of previous requests (i.e. a transaction-class workload, modeled as open arrivals in queuing models).

We emphasize that support for non-synthetic workload distributions (e.g., trace-driven load generation) can easily be added to the experimental setup. However, the focus of the present paper is on modeling of the server-side dynamics, and therefore, detailed characterization of the request patterns generated by the client side is beyond the scope of the present paper.

Performance experiments often iterate one or more parameters in the scenario. For instance, a series of experiments can be performed to study the effect of an increasing request rate on the mean response time of requests. We use scripts that iterate these parameters and instantiate workload scenarios templates using the parameter values.

3.3 Measuring performance

In this section we shortly describe how we obtain performance measurements of the ORBacus CORBA implementation using the Java Performance Monitoring Toolkit (JPMT) [4].

JPMT is based on event-driven monitoring. In general, two types of monitoring can be distinguished: time-driven monitoring and event-driven monitoring [8]. Time-driven monitoring observes the state of the monitored system at certain time intervals. This approach, also known as sampling, is often used to determine performance bottlenecks in software. For instance, by observing the call-stack every millisecond a list of methods using the most processing time can be obtained. Time-driven monitoring does not provide complete behavioral information, only snapshots. Event-driven monitoring is a monitoring technique where events in the system are observed. An event represents a unit of behavior, e.g., the creation of a new thread. Our monitoring toolkit implements the event-driven monitoring approach, since we require complete behavioral information, not just snapshots.

JPMT represents the execution behavior of applications by event traces, in which each event represents the occurrence of some activity, such as a method invocation or the creation of a new thread of execution. JPMT's event traces are similar to call-trees, but in addition to method invocations they also contain other event types.

The following elements of Java's execution behavior can be monitored:

- *Threading*: Java applications can have multiple threads of execution. The creation and destruction of these threads is monitored. Each thread has its own event trace.
- *Thread synchronization and cooperation*: Java uses monitors [5] to implement thread synchronization (Java's synchronized primitive) and cooperation (Java's `wait()`, `notify()`, `notifyAll()` methods in the `Object` class). JPMT can report how long a thread has contended for a monitor, how long it held a monitor, how long it spend waiting for a signal from another thread (cooperation), etc.
- *Method invocation*: The sequence of method invocations is represented in a call-tree notation for each thread.
- *Dynamic object allocation*: Object allocation and release monitoring can be used to track down excessive object allocation and release, which can cause performance problems, not only because of the allocation and release cost, but also because it may trigger garbage collection cycles.

Events are annotated with high-resolution timestamps, and depending on the event type other attributes such as used CPU time.

The user can select the parts of the application that should be monitored, by using filter mechanisms and specifying the event types of interest. There are filtering mechanisms to include or exclude threads and Java packages, classes, and methods.

The toolkit is not bound to a particular GUI or binary event trace file format – the event traces are accessible via an event trace API, which makes it possible to develop custom tools and scripts to process the monitoring results. Furthermore, the event traces can be represented in a human readable text file, which makes it possible to process the event traces without using the API. The instrumentation is added at run-time transparently to the user, and does not require availability of the source code. This allows the toolkit to be used for monitoring applications that come without source code.

The event traces produced by JPMT provide a lot of performance information. We developed post-processing scripts to make sense of this information. The post-processing scripts traverse the event traces and summarize the measurements. The CORBA experiment report produced by the post-processing scripts includes the following information:

- For the monitored parts of the receiver thread, the dispatcher thread, and the object implementation (servant) the completion times of each request are calculated. Also, the mean completion time, the variance, and a plot of the distribution of the completion time are reported. The post-processing script also reports how the completion times are made up from CPU time usage, garbage collection, and time spent waiting for locks.
- The queuing times of the FIFO queue of the dispatcher thread pool.
- Overall mean completion time and CPU time.
- The description of the client and server scenarios, used by the workload generator.
- Mean CPU usage on the server during the experiment.

3.4 Performance results

To assess the accuracy of the middleware performance model we have performed numerous experiments, both in the lab environment and with sumulation. For compactness of presentation a brief outline of the results is discussed below. In particular we present the results for two relevant scenarios. Scenario A represents a middleware system with a relatively large thread pool and relatively large object service time. Scenario B represents a system with a small thread pool and moderate object service time.

3.4.1 Scenario A

In scenario A we configure the CPU service demand of the target object to be 8 milliseconds, exponentially distributed. 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 varies with each experiment we conduct using this scenario.

Table 1 contains a summary of the performance measurement results, together with simulation results (discussed in section 4). The presented values are all averaged over the 15000 requests. The measured queuing time represents the time that requests are queued at the dispatcher thread pool. The measured completion time equals the (average) time between the arrival at the receiver thread and departure of a request from the dispatcher thread, after processing the request. The measured CPU time is the sum of all CPU times required for handling a request, including 1st and 2nd phase CPU times. It includes the 8 millisecond CPU time (on average) taken by the target object. The measured CPU utilization equals the fraction of time that the CPU was busy with executing requests.

As a sanity-check for the measured results note that for each experiment the average measured CPU utilization approximately equals the request arrival rate times the average measured CPU time.

The measured queuing and response times for this scenario are illustrated by Figure 4, together with simulated values.

3.4.2 Scenario B

In scenario B we configure the CPU service demand of the target object to be 5 milliseconds, exponentially distributed. The server ORB dispatcher thread pool has 4 threads. The client ORB will generate a workload of 15000 requests using a Poisson process. The arrival-rate varies with each experiment we conduct using this scenario.

Table 2 contains a summary of the performance measurement results, together with simulation results (discussed in section 4). The measured queuing and response times for this scenario are illustrated by Figure 5, together with simulated values.

4. SIMULATION AND VALIDATION

In this section we shortly describe how our performance model has been implemented in a simulation tool. After which we present simulation results for the scenarios outlined in the previous section. Finally, we discuss the validation of the model.

4.1 Model implementation

We have implemented the performance model described in section 2 in the Extend [9] simulation tool. The Extend tool can be used to implement models without using an advanced simulation programming language.

Models are constructed by connecting pre-built model building

Table 1. Scenario A experimental (measured) and simulation results

Arrival rate (requests/ms)	*Measured queuing time (ms)*	*Simulated queuing time (ms)*	*Measured completion time (ms)*	*Simulated completion time (ms)*	*Measured CPU time (ms)*	*Measured CPU utilization (percent)*	*Calculated CPU utilization (percent)*	*Simulated CPU utilization (percent)*
0.01	0.18	0	9.97	10.18	9.35	10.08	9.36	9.22
0.02	0.34	0	10.54	11.19	9.20	19.41	18.40	18.26
0.03	0.40	0	11.02	12.44	9.05	28.66	27.18	27.36
0.04	0.64	0.0002	12.79	14.60	9.27	37.77	37.10	37.05
0.05	0.83	0.01	14.77	17.24	9.24	46.48	46.24	45.79
0.06	1.04	0.03	18.56	21.27	9.26	56.06	55.61	55.88
0.07	1.47	0.28	22.71	26.36	9.30	63.66	65.11	64.86
0.08	2.06	1.50	29.24	34.85	9.26	72.81	74.15	73.72
0.09	4.74	4.76	38.87	49.36	9.25	80.61	83.29	82.56

Table 2. Scenario B experimental (measured) and simulation results

Arrival rate (requests/ms)	*Measured queuing time (ms)*	*Simulated queuing time (ms)*	*Measured completion time (ms)*	*Simulated completion time (ms)*	*Measured CPU time (ms)*	*Measured CPU utilization (percent)*	*Calculated CPU utilization (percent)*	*Simulated CPU utilization (percent)*
0.01	0.13	0	8.08	6.64	6.03	6.86	6.03	6.24
0.02	0.22	0.0006	8.68	7.02	6.00	13.41	12.01	12.32
0.03	0.28	0.0048	9.20	7.55	5.96	19.76	17.90	18.57
0.04	0.37	0.01	9.85	8.09	5.93	25.68	23.75	24.57
0.05	0.52	0.07	10.74	9.00	5.93	31.85	29.70	31.06
0.06	0.80	0.12	12.13	9.90	5.97	38.17	35.88	37.32
0.07	1.00	0.33	13.21	10.90	5.96	44.55	41.74	42.98
0.08	1.60	0.56	14.81	12.01	5.94	48.41	47.60	49.37
0.09	2.42	1.35	17.21	14.63	5.99	54.92	53.94	56.84
0.10	3.17	1.77	19.18	15.75	5.94	60.01	59.42	62.13
0.11	5.10	3.92	22.87	19.72	5.93	65.05	65.32	69.27
0.12	8.35	7.02	28.29	24.19	5.93	70.54	71.23	74.56
0.13	14.77	10.26	36.27	28.80	5.90	74.01	76.77	80.55

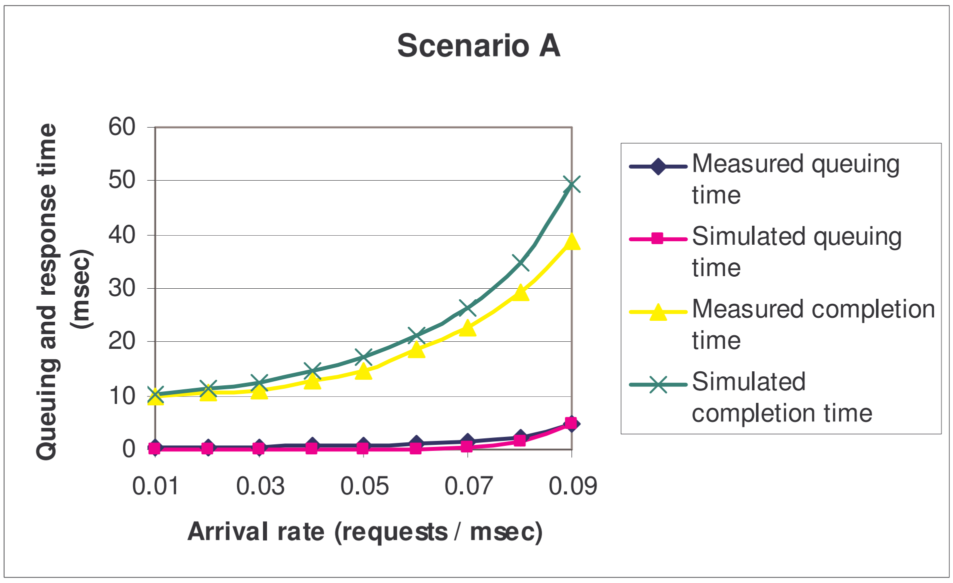


Figure 4. Experimental and simulation results for scenario A.

blocks together on a grid, using the Extend GUI. The user is not restricted to the pre-built building blocks. It is possible to add new building blocks, using the built-in simulation language *ModL*. Extend has a large library with pre-built building blocks, for instance blocks that implement the FIFO (first-in first-out) and PS (processor sharing) service disciplines. Extend models can be expressed in hierarchical manner, where blocks implement sub-models. Each block can have various parameters. The model parameter values can be specified in the notebook, for instance the inter-arrival times between requests, and the number of threads in the dispatcher thread pool.

For generating simulation results, Extend offers various building blocks to calculate results and report graphs. Using these blocks the model implementer can retrieve the required performance measures from the model, such as response times, throughput, queuing times, and resource utilizations.

4.2 Model simulation results

For scenario A and B we used the following parameter instances for the simulation runs:

Parameter	*Scenario A*	*Scenario B*
N_{con}	1	1
N_{poa}	1	1
N_{ser_1}	1	1
T_{disp}	10	4
T_{poa_1}	∞	∞
$T_{ser\_i,j}$	∞	∞
$\lambda_{1,1,1}$	*See remark 1*	
b_{rec}	*See remark 2*	
b_{poa}		
$b_{ser\_i,j}$	8 msec	5 msec
$b_{sern\_i,j}$	0 msec	0 msec
$b_{mar,\ ij}$	*See remark 2*	

Remarks:

1. For both scenarios the request arrival rate $\lambda_{1,1,1}$ is varied over a range of values, as indicated in the results.
2. In both scenarios we used the measured receiver thread, POA and marshaling CPU times from the experimental results as input for the model parameters. The aggregate measured CPU times are listed in Table 1 and Table 2.

4.3 Model validation

To assess the validity of the performance model, we compare the results from the performance experiments in section 3 with the simulation results.

4.3.1 Comparison of experimental and simulation results

Figure 4 and Figure 5 show that for both scenarios the completion and queueing time predictions from the simulation runs are accurate, compared to the measured completion and queueing times. This demonstrates the validity of the middleware performance model presented in section 2. Nonetheless, Figure 4 and Figure 5 illustrate that the model can be further refined. First of all, the difference between 1st phase and 2nd phase CPU time are ignored in the model. In the model a request is completed when both the 1st and the 2nd phase are processed by the CPU, while in practice the request is completed after the 1st phase. Second, only CPU times are considered in the model, while in the experiments I/O times will also (slightly) increase the measured completion times. We expect that the impact of each of those (and possibly more) differences between experiments and simulations

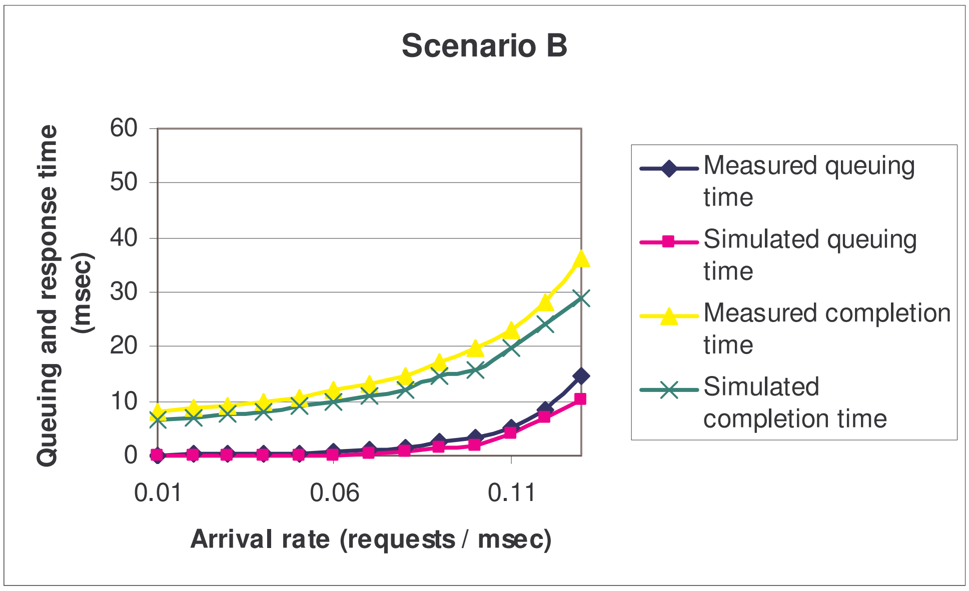


Figure 5. Experimental and simulation results for scenario B.

will be small, but nevertheless their aggregate impact will explain differences as seen in Figures 4 and 5.

4.3.2 Comparison of experimental and simulation run-times

One of the advantages of applying performance models, as compared to experimental performance analysis, is that model-based analysis is less time consuming. For the scenarios that we investigated we found that each simulation-run of 15000 requests took approximately between 2 and 3 minutes on a Pentium III 700 MHz (including post-processing of data into statistics). Using the lab set-up described in section 3, the experiment-runs took longer. Depending on the configured request arrival rate the simulations were up to 10 times faster than the experiments (excluding post-processing). The post-processing of experiment results typically takes around 9 minutes on the client machine in our lab setup, depending on the amount of raw monitoring data that has been generated.

5. CONCLUSIONS

In this paper we developed a basic quantitative performance model for CORBA-based middleware. 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 processing times of the middleware (including for instance the overhead induced by de-multiplexing the request to the proper application object, and the (un-)marshaling costs) and the processing times of the application. We have validated the model by comparing results from lab experiments with simulation results for a number of workload scenarios. The comparison of results from experiments and simulation looks promising. The results demonstrate that the performance predictions based on the model match accurately with the results from the lab experiments.

The results presented in this paper are a significant first step and indicate that the model-based approach for predicting middleware performance is very promising. Still, the results raise a number of challenges for further research. First, we plan to refine the performance model with aspects of CORBA that are currently not modeled. These aspects include other threading models than 'thread pool' and locking contention that may occur when the ORB is heavily loaded. We also plan to adjust the model to include a distinction between the 1st and 2nd phase of request processing in the receiver and dispatcher threads.

In addition to future work on model refinements, we will start using the performance model for optimizing the configuration of middleware implementations (e.g. optimal threading policy, optimal settings for dispatcher thread pool size, etcetera). Further, we will expand the focus of our research from server side middleware performance towards end-to-end application performance.

6. ACKNOWLEDGMENTS

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 Senter. Information on the EQUANET project is available from http://equanet.cs.utwente.nl/.

7. REFERENCES

[1] I. Abdul-Fatah, S. Majumdar, *Performance Comparison of Architectures for Client-Server Interactions in CORBA*, Proc. of the 18th International Conference on Distributed Computing Systems, May, 1998.

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

[3] R. Hariharan, W.K. Ehrlich, P.K. Reeser and R.D. van der Mei. *Performance of Web servers in a distributed computing environment.* In: Teletraffic Engineering in the Internet Era (eds. J. Moreira de Souza, N.L.S. da Fonseca and E.D. de Souza e Silva), Proceedings ITC17, 137-148, December 2001.

[4] 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.

[5] C.A.R. Hoare, *Monitors: An Operating System Structuring Concept*, Comm. ACM 17, 10:549-557 (October), 1974.

[6] M. Henning, S. Vinoski, *Advanced CORBA Programming with C++*, ISBN 0201379279, Addison-Wesley, 1999.

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

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

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

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

[11] C.M. Llado, P.G. Harrison, *Performance Evaluation of an Enterprise JavaBean Server Implementation*, Proc. of WOSP 2000, Ontario, Canada, 2000.

[12] 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.

[13] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, revision 2.5, OMG document formal/2001-09-01, 2001.

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

[15] F. Sheikh, J. Rolia, P. Garg, S. Frolund, A. Shepherd, *Layered Performance Modelling of a CORBA-based Distributed Application Design*, Proc. of the 4th International Conference on Analytical and Numerical Modeling Techniques with Application to Quality of Service modeling, Singapore, September, 1997.