

Performance Prediction of J2EE Applications using Messaging Protocols

Yan Liu, Ian Gorton

National ICT Australia (NICTA),
1430, NSW, Australia
{jenny.liu, ian.gorton}@nicta.com.au

Abstract. Predicting the performance of component-based applications is difficult due to the complexity of the underlying component technology. This problem is exacerbated when a messaging protocol is introduced to create a loosely coupled software architecture. Messaging uses asynchronous communication, and must address quality of service issues such as message persistence and flow control. In this paper, we present an approach to predicting the performance of Java 2 Enterprise Edition (J2EE) applications using messaging services. The prediction is done during application design, without access to the application implementation. This is achieved by modeling the interactions among J2EE and messaging components using queuing network models, calibrating the performance model with architecture attributes associated with these components, and populating the model parameters using a lightweight, application-independent benchmark. Benchmarking avoids the need for prototype testing in order to obtain the value of model parameters, and thus reduces the performance prediction effort. A case study is carried out to predict the performance of a J2EE application with asynchronous communication. Analysis of the resulting predictions shows the error is within 15%.

1 Introduction

Many software component models utilize synchronous communication protocols, such as Enterprise JavaBeans (EJB) based on RMI, and RPC-based CORBA or COM+ components. Synchronous communications dictates that the client process blocks until the response to its request arrives.

More loosely coupled software architectures can be constructed using asynchronous invocations. These place an intermediary messaging service between the client and server, decoupling their execution. In addition, asynchronous invocations are desirable for applications with high performance and scalability requirements. For these reasons, component technologies have been integrated with messaging protocols to support the development of applications with asynchronous architectures.

Messaging services are implemented by message-oriented middleware (MOM), such as Microsoft MSMQ, IBM WebSphere MQ, CORBA Notification Services and Sun's JMS (Java Messaging Service). JMS is a Java interface specification, which provides a standard way for Java applications to access enterprise messaging infra-

structure. MOM typically supports two forms of messaging: point-to-point (PTP) and publish/subscribe (Pub/Sub). In the PTP model, the message producer posts a message to a *queue*, and the message consumer retrieves the message from the queue. In the Pub/Sub model, a message producer publishes a message to a *topic*, and all consumers subscribing to the same topic retrieve a copy of the message. MOMs also define a set of reliability attributes for messaging, including non-persistent or persistent and non-transactional or transaction queues [18].

A component-based application using messaging protocols hence exploits an asynchronous, queue-based communication paradigm. It must also address additional architectural considerations such as the topology of component connections, message persistence and flow control. All of these factors can heavily influence the resulting application's performance [18].

However, the choice of application architecture must to be made early in the application life cycle, long before substantial coding takes place. Unwise decisions at design-time are often very difficult to alter, and could make it impossible to achieve the required performance level once the system has been delivered [5][6]. Consequently, the designer needs to be able to predict the performance of asynchronous components, working from an abstract design but without access to a complete implementation of the application.

Our previous work in [9] develops an approach to predicting the performance of only synchronous J2EE applications from design-level descriptions. The contribution of this paper is the extension of our approach to predict the performance of applications comprising both synchronous and asynchronous communications. This is achieved by modeling the component infrastructure that implements the messaging service. We then execute benchmarks to obtain values of model parameters associated with the performance characteristics of the underlying component infrastructure and the messaging service. We validate our approach through a case study, in which we compare predicted versus actual performance of an example application.

2 Related Work

Our previous work in [9] integrates analytical modeling and benchmark testing to predict the performance of J2EE applications using EJB components. A case study showed that without access the application source code, prediction can be accurate enough (prediction error is below 15%) to evaluate an architecture design. However, this work only addresses synchronous communication between components.

Performance modeling is a useful approach for performance analysis [16]. Traditional performance modeling techniques can be manually applied to applications based on Message-Oriented Middleware (MOM). [17] analyzes a multilayered queue network that models the communication between clients and servers via synchronous and asynchronous messages. [11] applies a layered QNM for business process integration middleware and compares the performance for both synchronous and asynchronous architectures. However, explicit values for performance parameters are required to solve these models, such as the CPU time that each operation is expected to use.

However, such performance parameters cannot be accurately estimated during an application design. A common practice therefore is to build a prototype and use this to obtain measures for the values of parameters in the model. For a complex application, this is expensive and time-consuming. Progress has been made to reduce the prototyping effort with tool support for automatic generation of test beds [1][3]. Although prototype testing can produce empirical evidence of the suitability of an architecture design, it is inherently inefficient in predicting performance as the application architecture inevitably evolves. Under change, the test bed has to be regenerated and redeployed, and the measurement has to be repeated for each change.

In related research towards software performance engineering, many approaches translate architecture designs mostly in United Modeling Language (UML) to analytical models, such as Queuing Network models [7], stochastic Petri nets [14] or stochastic process algebras [2]. In these approaches, the application workflow is presented in a sequence or state diagram, and a deployment diagram is used to describe the hardware and software resources, their topology and characteristics.

Importantly however, the component infrastructure and its performance properties are not explicitly modeled. These approaches therefore generally ignore or greatly simplify the details of the underlying component infrastructure performance. As a result, the models are rather inaccurate or non-representative. [8] developed a simulated model of CORBA middleware but the work is specific to threading structure of a CORBA server. Hence, little work has been done to develop an engineering approach to predict the performance of messaging applications during application design.

3 Major Performance Factors of J2EE Applications

J2EE includes several different component types, including EJB. EJB components act as servers and execute within a component container. A request to an EJB is passed through a method invocation chain implemented by the container and finally reaches the EJB method specified in the request. The invocation chain is used by the container to call security and transaction services that the EJB methods specify.

The container provides the hosting environment for EJBs and manages their lifecycle according to the context information of the request. The container also coordinates the interaction between EJBs and other J2EE services and facilities access to external data source connection pools.

To improve performance and scalability, the container is multi-threaded and can service multiple simultaneous EJB requests. Multiple instances of threads, EJBs and database connections are pooled to provide efficient resource usage in the container. Incoming requests are queued and wait for a container thread if none are available from the fixed size thread pool.

Concurrent EJB requests experience contention at three points inside the container. These are during request dispatching to an EJB, during container processing and during access to external data sources. As a result, apart from the underlying hardware and software environment, the performance of a deployed J2EE application depends on a combination of the following factors:

- the behavior of its application-specific EJB components and their interactions;
- the particular implementation of the component infrastructure, or container;
- the selected configuration settings for the container (e.g. thread pool size);
- the attribute settings of both the application components (e.g. the persistence attribute of EJBs) and the infrastructure components (e.g. the transaction isolation level of the container);
- the simultaneous request load experienced at any given time by the application [5].

Integrating a JMS messaging service with EJB components introduces further performance considerations. These include the topology of component connections, message persistence needs and flow control. For instance, non-persistent messaging has better performance than persistent messaging [18]. However persistent messaging creates an application that is guaranteed not to lose messages, and hence is more reliable. For an architect, the ability to quantify this performance/reliability trade-off without building each solution is desirable, as is determination of the level of performance that the overall system provides under load.

4. The Performance Prediction Approach

A performance prediction approach for J2EE applications with messaging protocol needs to encompass the following three aspects.

First, the performance model should explicitly represent the component container, the MOM service and their communication with application components.

Second, the service time of a request depends on the container and MOM attributes. For example, in MOM-based applications, the setting of a messaging attribute is an architectural design decision and the effect on performance should be modeled as a function of the messaging attributes of interest.

Third, an application-independent performance profile of the container and the MOM infrastructure is required. This is because the container and message server implementation and the operating system/hardware platform must be taken into account to be able to make accurate application performance predictions.

The relationship between the performance model and the component container performance profile for a selected architecture model are represented as a performance prediction framework in [9]. In this framework, a queueing network model (QNM) P models the component infrastructure by identifying the main components of the system, and noting where queuing delays occur.

An architect has several alternatives to fulfill the same functionality using EJB technology. For example, a server side EJB component can be made either stateless or stateful, simply by setting an attribute. Each architecture alternative impacts the service time of the component container. Therefore the component architecture model f^A is a function of the service time of the components participating in an architecture A . The output of f^A is the input to P .

Performance profiles are required to solve the parameter values of the performance model. They are obtained from benchmarking measurements. The benchmark application differs from an application prototype in that the business logic of the benchmark

is much simpler than a prototype. The operations of the benchmark are designed simply to exercise performance sensitive elements of the underlying component container. The aim is to determine the performance profile of the container itself, and not to evaluate the overall application performance (the traditional aim of benchmarking). By using a simple benchmark application, we can remove any unpredictability in performance due to application business logic.

The model is finally populated using the performance profile and used for performance prediction. This approach enables performance prediction during the design of software applications that are based on a specific component technology.

A comprehensive description of this approach can be found in [9][10]. It is designed to support the following use cases during performance engineering:

- Support efficient performance prediction under architecture changes where components are added or modified.
- Capacity planning of the system, such as predicting the average response time, throughput and resource utilization under the expected workload.
- Reveal performance bottlenecks by giving insight into possible flaws in architecture designs.

The requirements of this approach are:

- Ensuring a reasonable level of accuracy for performance prediction. According to [12] (page 116), prediction error within 30% is acceptable.
- Cost effective. The approach must be faster than prototyping and testing.

5. The Performance Model

5.1. The QNM of a J2EE container using JMS Queues

A performance model should capture the component container behavior when processing a request from a client. For this reason, we focus on the behavior of the container in processing EJB method invocation requests. As EJB containers process multiple simultaneous requests, the threading model utilized must also be represented. The QNM in **Fig. 1** models the main infrastructure components involved and their interactions.

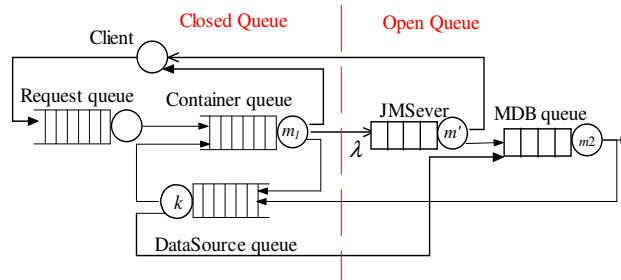


Fig. 1. The QNM model of a J2EE server with a JMS Queue

The model comprises two sub-networks, a closed and an open QNM. A closed QNM is appropriate for components using synchronous communication, as component containers employ a finite thread pool that effectively limits the maximum requests active in the server. An open QNM models asynchronous communication as a component container sends a message to a JMS queue, and the message is forwarded to a message driven bean (MDB) to process the business logic.

In the closed model, application clients represent the ‘proxy clients’¹ (such as servlets in a web server) of the EJB container. Consequently, a client is considered as a delay resource and its service time equals the *thinking* time between two successive requests. A request to the EJB container is interpreted and dispatched to an active container thread by the request handler. The request handler is modeled as a single server queue with no-load dependency. It is annotated as *Request queue* in the QNM.

The container is multi-threaded, and therefore it is modeled as a multi-server *queue* with the thread capacity m_l and no load dependency. It is annotated as *Container* in the QNM.

The database clients are in fact the EJBs that handle the client request. Database access is therefore modeled as a delay server with load dependency. Its active database connections are denoted as k in the QNM, and the operation time at the database tier contributes to the service demand of the *DataSource* queue.

In the open model, asynchronous transactions are forwarded by the container to a queue managed by a JMS server. The JMS server is multi-threaded, and has a threshold for flow control to specify the maximum number of the messages pending in the JMS server. Assuming that the arrival rate of requests is a Poisson distribution with rate λ requests per second and the service time is exponential, we can model the JMS server as an $M/M/m/W$ queue, where m' is the number of JMS server threads and W is its flow control threshold.

A message is subsequently removed from the queue by a MDB instance, which implements the business logic. MDBs are asynchronous message-handling façades for data access carried out in entity beans. MDB instances are also managed by the EJB container and are associated with a dedicated server thread pool. So the *MDB* queue is modeled as a load-independent multi-server queue.

The implementation of an EJB container and JMS server is complex and vendor specific. This makes it extremely difficult to develop a performance model that covers all the relevant implementation-dependent features, especially as the EJB container source code is not available. For this reason, our quantitative model only covers the major factors that impact the performance of applications, and ignores many other factors that are less performance sensitive. Specifically, we do not currently consider workloads that include large data transfers. As a result, the network traffic is ignored and the database contention level is reduced.

¹ As opposed to clients driven by human interaction, proxy clients such as servlets continually handle requests that arrive at a web server.

5.2. The Architecture Model

The task of solving the QNM in Fig. 1 involves obtaining the service demand of each queue. We need to calibrate the component container that will host the alternative designs in order to obtain the service demands of each request on each queue.

The service demand of the *Request* queue equals the service time of a request being accepted by the server and dispatched to the *Container* queue. It can thus be considered as a constant. The *Container*, *DataSource*, *JMS* and *MDB* queues are responsible for processing the business logic and this comprises the majority of the service demands on these queues.

Fig. 2 shows the state diagram for processing transactions in an EJB container. The container has a set of operations that a request must pass through, such as initializing a set of container management services, invoking the generated skeleton code for a bean instance, registering a transaction context with the transaction manager, finalizing a transaction and reclaiming resources.

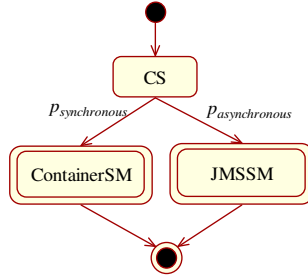


Fig. 2. Overall state diagram of the J2EE server

These container operations are identical for all requests, and the service time can again be considered constant, denoted as T_0 . For convenience, these states as a whole are referred to as a composite, CS. Synchronous transactions are processed by the *Container* queue, modeled as a compound state machine *ContainerSM* with a probability $p_{synchronous}$, while asynchronous messages are posted to the JMS server, modeled as a compound state machine *JMSSM* with a probability $p_{asynchronous}$. *ContainerSM* and *JMSSM* are further decomposed into models of different component architectures. The service times of operations in *Container*, *DataSource*, *JMS* and *MDB* queue are modeled as f_c , f_d , f_j and f_m respectively.

From the above analysis, we know that f_c and f_d are determined by the component architecture in *ContainerSM* (e.g. optimizing data access to an entity bean by different cache management schemes). The comprehensive architecture models are developed in [9][10]. The models for container managed persistence of entity beans are listed below as an example:

$$f_c = hT_1 + (1 - h)T_2 \quad (1)$$

$$f_d = T_{find} + T_{load} + pT_{store} \quad (2)$$

Where

- h is the entity cache hit ratio;
- p is the ratio of operations updating entity data;
- T_1 is the service time for the container to access the entity data in its cache;
- T_2 is the service time of the container to load/store an entity bean instance from/to disk;
- T_{find} is the service time of identifying the entity instance by its primary key;
- T_{load} is the service time of loading data from the database into the entity bean cache;
- T_{store} is the service time of storing updates of an entity bean data.

For each business transaction, the necessary behavioral information can be extracted from design descriptions, such as use case diagram, and sequence diagrams [9][10]. This must capture the number and type of EJBs participating in the transaction, as well as the component architecture selected. Therefore for each transaction r the service demand at the *Container* and *DataSource* queue can be calculated as

$$D_{r,Container} = T_0 + \sum_{each-bean} f_c \quad (3)$$

$$D_{r,DataSource} = \sum_{each-bean} f_d \quad (4)$$

The execution demands of a message façade for entity beans in the *MDB* queue is exactly the same as the session façade for entity beans in the *Container* queue. Therefore the service demands of the *MDB* queue can be modeled in the same way, namely:

$$D_{r,MDB} = T_0 + \sum_{each-bean} f_c \quad (5)$$

In this paper we assume the JMS server is co-located in the same container with the EJB components to minimize communications overheads. The JMS queue used to communicate between components can be configured as persistent or non-persistent, depending on the reliability requirement of the application. In this paper, we consider the following messaging attributes:

- Non-persistent messages: Messages are not persisted in the JMS.
- Persistent messages with *cache-flush*: A message is first copied into memory and then flushed to the disk. The transaction cannot be committed until all of writes have been flushed to disk.
- Persistent messages with disabled *cache-flush*: *Disabled* means that transactions complete as soon as file store writes are cached in memory, instead of waiting for the writes to successfully reach the disk.

Fig. 3 shows the *JMSSM* decomposition for these messaging attributes. For non-persistent messages, the message is delivered directly. For persistent messages with the *cache-flush* setting, the message is copied to memory and then written to a file store. The XA transaction manager forces a two-phase commit if there is more than one transaction participant.

When *cache-flush* is *disabled*, the transaction ends once the message is cached and the message is persisted to the file store in a background thread. The following parameters in Table 1 are necessary for calculating the *JMSSM* architecture model in equation (6)-(8) for three messaging attributes.

$$f_j^{non-persistent} = T_{Jsend} \quad (6)$$

$$f_j^{cache-flush} = T_{Jcache} + T_{Jwrite} + T_{Jxa} + T_{Jsend} + T_{jremove} \quad (7)$$

$$f_j^{disabled} = T_{Jcache} + T_{Jxa} + T_{Jsend} \quad (8)$$

Note that $f_j^{disabled}$ is calculated along the branch consuming the least service time in Fig. 3 (c). The service demand of a JMS server queue with a specific messaging attribute is

$$D_{jms} = f_j \quad (9)$$

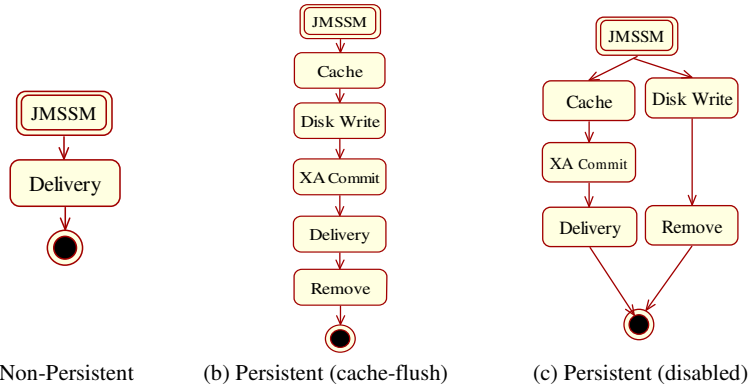


Fig. 3. The JMS sub-state machine decomposition

Table 1. Parameters of JMSSM architecture models

T_{Jcache}	The service time of copying a message into memory
T_{Jxa}	The service time of an XA transaction commit
T_{Jwrite}	The service time of writing a message to disk
T_{Jsend}	The service time of sending a message to its destination
$T_{Jremove}$	The service time of removing the message from its persisted list

6. Benchmarking

Benchmarking is the process of running a specific program or workload on a machine or system and measuring the resulting performance [15]. In our approach, benchmarking is used to provide values for certain parameters in the performance models that are used for prediction.

6.1. The benchmark design and implementation

The key innovation of our modeling method is that we model the behavior solely of component infrastructure itself, and augment this with architecture and application-specific behavior extracted from the application design. This implies that the bench-

mark application should have the minimum application layer possible to exercise the component infrastructure in a meaningful and useful way. Component technologies leverage many standard services to support application development. The benchmark scenario is thus designed to exercise the key elements of a component infrastructure involved in the application execution.

We have designed and implemented a benchmark suite to profile the performance of EJB-based applications on a specified J2EE implementation. The implementation of the benchmark involves a session bean and an entity bean. The benchmark scenario is shown in Fig. 4. In order to explore the JMS server and MDB container’s performance profiles, a new scenario is introduced in Fig. 5, which involves a JMS queue and a MDB object. Table 2. lists the J2EE services and design patterns used by each transaction in the benchmark application. The benchmark scenario can easily be extended for other performance profiles, such as creating/removing an EJB instance.

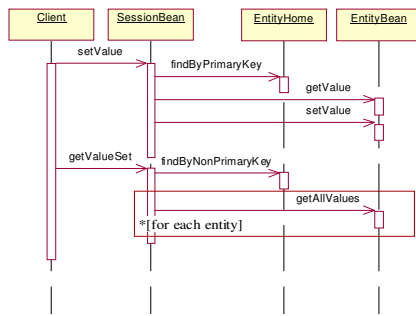


Fig. 4. The basic benchmark scenario

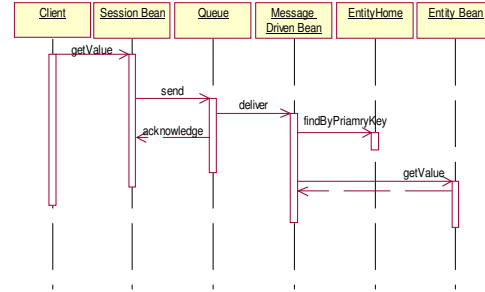


Fig. 5. The benchmark scenario with JMS and MDB

Table 2. J2EE service and design pattern usage matrix

Transaction	J2EE service	Design Pattern
<i>setValue</i>	EJB 2.0 (stateless session bean, entity bean with container managed persistence), container managed transaction, JNDI, security	Session façade, service locator, read mostly
<i>getValueSet</i>		Session façade, service locator, read mostly, aggressive loading
<i>getValue</i>	Above services and JMS, MDB	Message façade, service locator, read mostly

The benchmark suite also comprises a workload generator, monitoring utility and profiling toolkit. A monitoring utility is implemented using the Java Management Extensions (JMX) API. It collects performance metrics for the application server and the EJB container at runtime, for example the number of active server threads, active database connections and the hit ratio of the entity bean cache.

A profiling toolkit *OptimizeIt* [13] is also employed. *OptimizeIt* obtains profiling data from the Java virtual machine, and helps in tracing the execution path and collecting statistics such as the percentage of time spent on a method invocation, from which we can estimate the percentage of time spent on a key subsystems of the J2EE server infrastructure. Profiling tools are necessary for black box COTS component systems, as instrumentation of the source code is not possible.

The benchmark clients simulate requests from proxy applications, such as servlets executing in a web server. Under heavy workloads, this kind of proxy client has an ignorable interval between two successive requests. Its population in a steady state is consequently bounded². Hence the benchmark client spawns a fixed number of threads for each test. Each thread submits a new service request immediately after the results are returned from the previous request to the EJB. The ‘thinking time’ of the client is thus effectively zero. The benchmark also uses utility programs to collect the measurement of black-box metrics, such as response time and throughput.

6.2 Measurement

The benchmark suite must be deployed on the same platform (both software and hardware) as the target application. In our case study, the benchmarking environment consists of two machines, one for clients, and the other for application and database server. They are connected by a 100MB Ethernet LAN. The hardware and software configuration is listed in Table 3. For the application and database server machine, *HyperThreading* is enabled, effectively making four CPUs available. Two CPUs are allocated for the application server process and the other two CPUs are allocated for the database server process.

Table 3. Hardware and software configuration

Machine	Hardware	Software
Client	Pentium 4 CPU 2.80 GHz, 512M RAM	Windows XP Prof. BEA WebLogic server 8.1,
Application and database server	Xeon Dual Processors, 2.66 GHz, HyperThreading enabled, 2G RAM	WindowsXP Prof. BEA WebLogic server 8.1, JDK1.4 with settings <code>-hotspot</code> , <code>-Xms512m</code> and <code>-Xmx1024m</code> . Oracle 9i

Each experiment has three phases, *rampUp*, *steadyState* and *rampDown*. The system is started and initialized in the *rampUp* stage for 1 minute. The system then transfers to *steadyState* for 10 minutes. Each experiment is run several times to achieve high levels of confidence that the performance difference between two runs under the same initialization conditions is not significant (below 3%). The values of parameters obtained from benchmarking are listed in Table 5. They are used populate the performance model we developed above with specific hardware/software configuration in Table 3.

² A web server has configuration parameters to limit the active workload. For example, Apache uses *MaxClient* to control the maximum number of workers, thus the concurrent requests to the application server are bounded.

7. Case Study: Performance Prediction of Stock-Online

7.1 The Stock-Online Example

Stock-Online [4] is a simulation of an on-line stock-broking system. It models typical e-commerce application functions and aims to exercise the core features of a component container. Stock-Online supports six business transactions, two of which are relatively heavyweight, write intensive transactions, and four others that are lightweight read or write transactions. Its use case diagram is shown in Fig. 6. The supporting database has four tables to store details for accounts, stock items, holdings and transaction history. The application transaction mix can be configured to model a lightweight system with read-mostly operations, or a heavyweight one with intensive update operations. The default transaction mix is listed in Table 4.

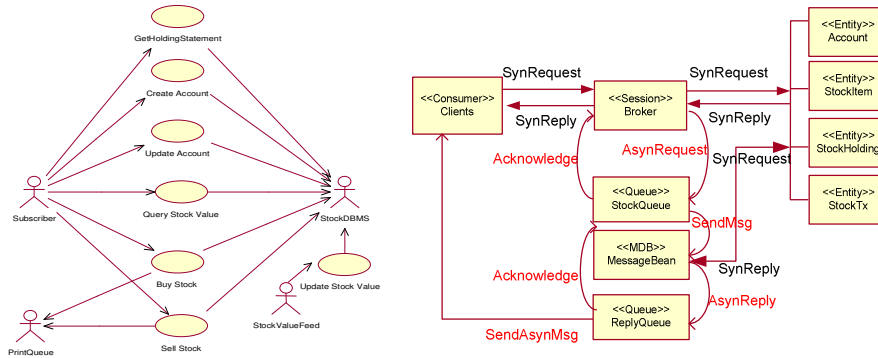


Fig. 6. Stock-Online use case

Table 4. Default Stock-Online business models

Usage Pattern	Transaction	Transaction Mix
Read-only point (read one record)	Query stock	70%
Read-only multiple (read a collection of records)	Get stockholding	12%
Read-write (inserting records, and updating records; all requiring transactional support)	Create account	2%
	Update account	2%
	Buy stock	7%
	Sell stock	7%

Suppose that Stock-Online is to be implemented using EJB components. A common solution architecture is to use Container Managed Persistence (CMP) entity beans, using the standard EJB design pattern of a session bean as a façade to the entity beans. A single session bean implements all transaction methods. Four entity beans, one each for the database tables, manage the database access.

Buy and *Sell* transactions are the most demanding in terms of performance, each involving all four entity beans. For this reason, in our implementations *Buy* and *Sell* transactions operate asynchronously³, while the remaining short-lived transactions are always called synchronously. In this asynchronous architecture, the session façade EJB sends a request as a message to a JMS queue and returns immediately to the client. The queue receives the message and triggers the *onMessage* method inside a MDB instance. *onMessage* embodies the business logic and accesses the entity beans for data access.

The Stock-Online architecture can be modeled using the QNM developed in Fig. 1, in which the stereotypes <<session>> and <<entity>> beans are included in the *Container* queue, <<MDB>> bean are included in the *MDB* queue and <<Queue>> is modeled as the *JMSServer* queue.

Table 5. Parameters from benchmarking

T_o	7.833
T_l	1.004
T_2	35.856
T_{create}	0.497
T_{load}	0.215
T_{store}	0.497
T_{insert}	0.497
T_{Jxa}	10.972
T_{Jsend}	0.929
$T_{Jwrite} + T_{Jremove}$	109.718
Cache hit ratio (<i>h</i>)	0.69

Table 6. Stock-Online service demands

<i>Transaction, Queue</i>	Service demand
<i>All, Request</i>	0.204
<i>NewAccount, Container</i>	34.726
<i>NewAccount, DataSource</i>	0.497
<i>UpdateAccount, Container</i>	35.777
<i>UpdateAccount, DataSource</i>	0.927
<i>BuyStock, MDB</i>	81.959
<i>BuyStock, DataSource</i>	2.780
<i>SellStock, MDB</i>	70.844
<i>SellStock, DataSource</i>	2.780
<i>QueryStockValue, Container</i>	19.641
<i>QueryStockValue, DataSource</i>	0.430
<i>GetStockHolding, Container</i>	76.708
<i>GetStockHolding, DataSource</i>	0.516
<i>BuyStock and SellStock/JMS (non-persistent)</i>	0.929
<i>BuyStock and SellStock/JMS (cache-flush)</i>	122.623
<i>BuyStock and SellStock/JMS (disabled)</i>	12.905

Assume Stock-Online is to be deployed on the platform described in Table 3. In order to predict its performance, we need to estimate the service demand of each transaction imposed on each queue in the QNM. As we have discussed, the necessary behavioral information of each transaction must be captured such as the number and type of EJBs participating in the transaction, as well as the component architecture

³ This is exactly how online stock broking sites such as E-Trade operate.

selected. For example, the transaction to query stock value has read-only access to one entity bean *StockItem*, and we use the performance profile in **Table 5** to populate Equation (10), hence its service demand on the *Container* queue can be calculated as:

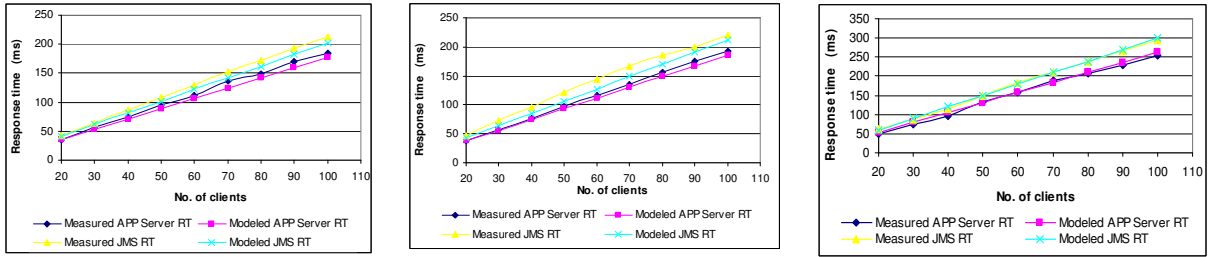
$$D_{QueryStock\ Value, Container} = T_0 + hT_1 + (1 - h)T_2 = 19.641 \quad (10)$$

Table 6 lists the Stock-Online's service demand for each transaction in each queue using equation (3)-(9).

7.2 Predicting performance for different messaging attributes

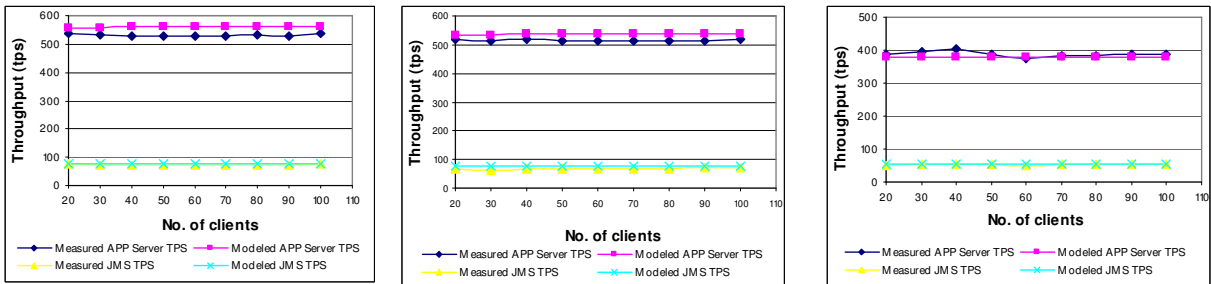
With the benchmark results, the performance model can be populated with performance profiles, and solved to predict the performance of Stock-Online with the three different messaging quality of service attributes. The arrival rate of requests for the open QNM depends on the throughput of the application server sending asynchronous *Buy/Sell* requests, while the load imposed on the *DataSource* queue by the open QNM affects the response time of the closed QNM. The overall model is solved iteratively between the two QNMs using the Mean Value Analysis (MVA) algorithm.

In order to validate the accuracy of these predictions, Stock-Online has been implemented and deployed. A workload generator injects six types of transactions into the Stock-Online application. The response time and throughput of both the EJB server and the JMS server are measured for a range of client loads.



(a) Non-persistent (b) Persistent-disabled (c) Persistent-cache-flush

Fig. 7. Response Time with three messaging attributes (measured vs. predicted)



(a) Non-persistent (b) Persistent-disabled (c) Persistent-cache-flush

Fig. 8. Throughput with three messaging attributes (measured vs. predicted)

Fig. 7 and Fig. 8 show the predictions for response time and throughput respectively. Most prediction errors are less than 10% and the maximum error is within 15%. The predictions demonstrate that the throughput achieved with the cache flush disabled option for persistent message is approximately 96% of that achieved by non-persistent messages. Hence these two architectures are comparable in terms of performance. This is verified by measurement of the Stock-Online implementation, where the cache disabled option provides 95% throughput of non-persistent messaging.

Persistent messages with *cache-flush* are much slower, because of disk write operations. We predict that the performance of *cache-flush* message persistency degrades approximately 28%. The actual measures show that the performance degradation is approximately 32%. Hence the case study demonstrates that our approach is accurate enough to support messaging architecture evaluation in terms of performance.

Buy and *Sell* transactions have higher service demands than the others transactions. Dispatching them to the JMS server reduces the load in the EJB server. Therefore the average application response time is smaller than the JMS server with MDB. This can also be observed from our predictions.

8. Conclusions

In this paper, we present an approach for predicting, during the design phase, the performance of component-based applications with both synchronous and asynchronous communications. It combines analytical models of the component behavior and benchmark measurements of a specific platform. The case study is a J2EE application using the JMS API. Our approach is applied to predict the performance of three different MOM QoS attributes and two entity bean cache architectures. Without access to the application source code, the error of prediction is below 15%.

Currently, predictions are carried out manually. However it is possible to automate much of the process by leveraging research in the software engineering community. For example, the benchmark application can be generated and run automatically using the tool developed by Cai et al. [1]. An on-going research project is to automatically generate the performance model and the benchmark suite from a design. Also more evidence is required that the approach is broadly applicable and scalable. To this end we are working to:

- Apply this approach to other middleware platforms, such as .NET and CORBA.
- Test the approach on more complex applications.
- Design software engineering tools that hide the complexity of the modeling and analysis steps in our performance prediction approach from an architect.

References

- [1] Cai, Y.; Grundy, J.; Hosking, J.: Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool, Proc. IEEE Int. Conf. on Automated Software Engineering (ASE), September, 2004.

- [2] Canevet, C.; Gilmore, S.; Hillston, J.; Prowse, M.; Stevens, P.: Performance modeling with UML and stochastic process algebras. *IEE Proc. Computers and Digital Techniques*, 150(2):107-120, 2003.
- [3] Denaro, G.; Polin, A.; Emmerich, W.: Early Performance Testing of Distributed Software Applications. Proc. Int. Workshop on Software and performance (WOSP), pp. 94–103, January 2004.
- [4] Gorton, I.: Enterprise Transaction Processing Systems, Addison-Wesley, 2000.
- [5] Gorton, I. and Liu, A.; Performance Evaluation Of Alternative Component Architectures For EJB Applications, *IEEE Internet Computing*, vol.7, no. 3,2003, pp.18-23.
- [6] Gorton, I.; Haack, J.: Architecting in the face of uncertainty: an experience report, Proc. 26th Int. Conf. on Software Engineering (ICSE), pp. 543- 551, 2004.
- [7] Gu, G. P.; Petriu, D. C.: XSLT transformation from UML models to LQN performance models, Proc. Int. Workshop on Software and performance (WOSP), pp. 227-234, 2002.
- [8] Harkema, M.; Gijzen B.M.M.; Mei, R.D.; Hoekstra, Y.: Middleware Performance: A Quantitative Modeling Approach, Proc. Int. Sym. Performance Evaluation of Computer and Communication Systems (SPECTS), 2004.
- [9] Liu, Y.; Fekete, A.; Gorton, I.: Predicting the performance of middleware-based applications at the design level, Proc. Int. Workshop on Performance and Software Engineering (WOSP), pp 166-170, 2004.
- [10] Liu, Y.: A Framework to Predict the Performance of Component-based Applications, PhD Thesis, University of Sydney, Australia, 2004.
- [11] Liu, T.K.; Behroozi, A.; Kumaran, S. A performance model for a business process integration middleware, *IEEE Int'l Conf. on E-Commerce*, 2003, pp. 191-198.
- [12] Menascé, D.A.; Almeida, V.A.F.; Capacity Planning for Web Performance, Metrics, Models, and Methods. Prentice-Hall, 1998.
- [13] OptimizeIt Suite, <http://www.borland.com/optimizeit/>
- [14] P. King and R. Pooley: Derivation of Petri Net Performance Models from UML Specifications of Communications Software, Proc. Int. Conf. on Tools and Techniques for Computer Performance Evaluation (TOOLS), 2000.
- [15] Saavedra, R. H., Smith, A. J.: Analysis of benchmark characteristics and benchmark performance prediction, *ACM Transactions on Computer System*, vol. 14, no. 4, pp. 344-384,1996.
- [16] Simeoni, M.; Inverardi, P.; Di Marco, A.; Balsamo, S. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp 295-310, 2004.
- [17] Sridhar R., Perros, H. G.: A multilayer client-server queueing network model with synchronous and asynchronous messages, *IEEE Trans. on Software Engineering*, vol. 26, no. 11, pp. 1086-1100, 2000.
- [18] Tran, P.; Gosper, J.; Gorton, I.: Evaluating the Sustained Performance of COTS-based Messaging Systems, in *Software Testing, Verification and Reliability*, vol 13, pp 229-240, 2003.